Scalable services for massively multiplayer online games

Maxime Véron

Université Pierre et Marie Curie

Doctoral Thesis

Scalable services for massively multiplayer online games

Author:
Maxime Véron

Jury members:
Petit Franck,
Fedak Gilles,
Pierson Jean-Marc,
Friedman Roy,
Sens Pierre,
Marin Olivier

Supervisors:
Pierre Sens, Olivier Marin,
Sébastien Monnet

25 September 2015

"Choose a job you love, and you will never have to work a day in your life."

Confucius

Abstract

Scalable services for massively multiplayer online games

by Maxime VÉRON

Massively Multi-player Online Games (MMOGs) aim at gathering an infinite number of players within the same virtual universe. Yet all existing MMOGs rely on centralized client/server architectures which impose a limit on the maximum number of players (avatars) and resources that can coexist in any given virtual universe [1]. This thesis aims at proposing solutions to improve the scalability of MMOGs.

There are many variants of MMOGs, like role playing games (MMORPGs), first person shooters (MMOFPSs), and battle arenas (MOBAs), each with a specific set of design concerns: low latency, artificial intelligence (AI) for the control of characters, expandable universe, so on and so forth. To address the wide variety of their concerns, MMOGs rely on independent services such as virtual world hosting, avatar storage, matchmaking, cheat detection, and game design. This thesis explores two services that are crucial to all MMOG variants: matchmaking and cheat detection. Both services are known bottlenecks, and yet current implementations remain centralized.

Matchmaking allows players to form teams and to find opponents. Current service implementations raise important issues regarding player experience. They often lead to mismatches where strong players face weak ones, and response times can be exceedingly long. This thesis conducts an extensive analysis over a large dataset of user traces in order to propose a set of matchmaking solutions that scale with the number of clients. Interestingly, some of these solutions are centralized and still deliver excellent service.

Current refereeing services for MMOGs also rely on centralized architectures, which limits both the size of the virtual world and the number of players. Conversely to matchmaking, cheat detection requires a high frequency of computations per player: therefore a distributed solution seems best for scalability purposes. This thesis shows that it is possible to design a peer to peer refereeing service on top of a reputation system. The resulting service remains highly efficient on a large scale, both in terms of performance and in terms of cheat prevention. Since refereeing is somewhat similar to failure detection, this thesis extends the proposed approach to monitor failures. The resulting failure detection service scales with the number of monitored nodes and tolerates jitter.

Résumé

Les jeux massivement multijoueurs en ligne(MMOG) visent à rassembler un nombre infini de joueurs dans le même univers virtuel. Pourtant, tous les MMOG existants reposent sur une architecture client / serveur centralisée qui impose une limite sur le nombre maximum de joueurs (avatars) et les ressources qui peuvent coexister dans l'univers virtuel [1]. Cette thèse vise à trouver des solutions en pour améliorer le passage à l'échelle des MMOGs.

Il existe de nombreuses variantes de MMOGs, comme les jeux de rôle (MMORPG), jeux de tir à la première personne (MMOFPSs), et arènes de combat (MOBAs), chacune avec un ensemble spécifique de spécifications: faible latence, intelligence artificielle (IA) pour le contrôle des personnages, univers extensible et ainsi de suite.

Pour répondre à la grande variété de leurs préoccupations, les MMOG comptent sur des services indépendants tels que l'hébergement de monde virtuel, le stockage d'avatar, le "matchmaking" (service de mise en relation), la détection de triche, et le "game design" (techniques de conception du jeu).

Cette thèse explore deux services qui sont essentiels à toutes les variantes de MMOG: matchmaking et la détection de triche. Ces deux services sont des goulots d'étranglement connus, et les implémentations actuelles sont encore centralisées.

Le matchmaking permet aux joueurs de former des équipes et de trouver des adversaires. Les implémentations de services actuels possèdent des problèmes importants en matière d'expérience de joueur. Ils conduisent souvent à des inadéquations où des joueurs de fort niveau sont confrontés à des débutants, et les temps de réponse peuvent être extrêmement longs. Cette thèse mène une analyse approfondie sur une grande base de données d'utilisateur afin de proposer un ensemble de solutions de matchmaking qui évoluent avec le nombre de clients. Fait intéressant, certaines de ces solutions sont centralisées et offrir pourtant un excellent service.

Les services d'arbitrage actuels pour MMOGs comptent également sur des architectures centralisées, ce qui limite à la fois la taille du monde virtuel et le nombre de joueurs. Inversement au matchmaking, la détection de triche nécessite une puissance élevée de calculs par joueur: une solution distribuée semble préférable à des fins d'évolutivité.

Cette thèse montre qu'il est possible de concevoir un réseau pair à pair pour le service d'arbitrage en utilisant un système de réputation. Le service résultant reste très efficace sur une grande échelle, à la fois en termes de performance et en termes de prévention de la fraude.

Comme l'arbitrage est un similaire à la détection de fautes, cette thèse étend l'approche proposée pour surveiller les défaillances. Le service de détection de défaillance résultant évolue avec le nombre de nœuds et résiste à la gigue.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisors and supervisors Pierre Sens, Olivier Marin and Sebastien Monnet for the continuous support of my Ph.D study and related research, for their patience, motivation, and insightful comments. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better advisors and supervisors for my Ph.D study.

I would like to thank the rest of my thesis committee: Prof. Jean-Marc Pierson and Dr. Gilles Fedak, for their insightful comments and encouragement. I would like also to thank the jury members Associate Prof. Roy Friedman and Prof. Franck Petit.

I thank my fellow labmates in for the stimulating discussions, and for all the fun we have had in the last three years. A particular thank goes to Thomas Preud'homme who made the office feel like a welcoming place when I arrived after my internship.

Last but not the least, I would like to thank my family: my parents, my sister and my fiancé for supporting me spiritually throughout this thesis and in my life in general.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

The popularity of multiplayer gaming began to grow in the 90's. Fifteen years later a lot of multiplayer games have switched to massively multiplayer architectures. These games gather immense playerbases. The most famous of them, World of Warcraft [1] peaks at twelve million users [3]. As seen in Figure 1.1, even though the popularity of this massively multiplayer online game (MMOG) had been declining over the past years, World of Warcraft managed to restore a big part of its playerbase. This demonstrates that MMOGs are still an important part of the gaming industry: the lowering popularity was due to a lack of quality content. A recent game called World of Tanks received a Guiness world record for the highest number of concurrent users (eight hundred thousand) on a single MMO server [4]. It is important to notice that World of Tanks scales up by relying on the nature of its game: slow paced, small groups of players, limited actions and immutable world.

MMOGs are games meant to be played online with a massive amount of players. They usually offer at least one persistent world. These games are available for most network-capable platforms, including personal computers, video game consoles, or smartphones and other mobile devices. MMOGs enable players to cooperate and compete with each other on a large scale, and sometimes to interact meaningfully with people around the world.

[1] http://battle.net/wow/

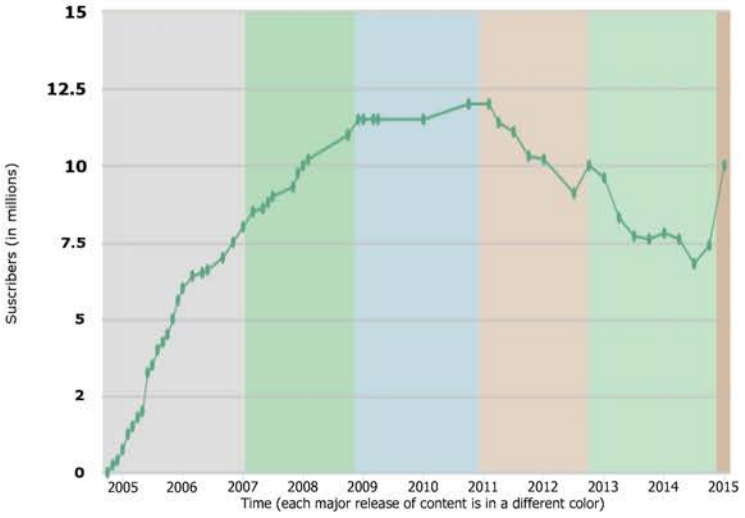


FIGURE 1.1: Popularity of World of Warcraft over the years

Good examples of those games are MMORPGs (MMO Role-Playing Game) like World Of Warcraft [2], Aion[3], and Tera [4]. Players connect to the universes of those games, and create their avatar: a representation of their own self in the virtual world. This avatar can perform actions that are defined by the video game designer. Depending on the nature of the game, those actions may vary.

Even recent, MMOGs fail to handle the load they receive in the early stages of the game's launch [5, 6] because of their high popularity. In order to handle such a big playerbase one straightforward idea is the peer to peer (P2P) model: the bigger the playerbase becomes, the more resources are available in the system. To do so, ingenious systems have to be built in order to manage efficiently an entire virtual environment such as an MMOG universe while offering as much reliability and consistency as a centralized server.

1.1 Current MMOGs limitations

Even though scalability of MMOGs virtual worlds is still an open issue, it has been vastly covered and addressed. Some other services, namely cheat detection and matchmaking, need improvement in order to achieve high scalability and I attempt to scale them in this thesis.

[2]http://battle.net/wow/
[3]http://www.aionfreetoplay.com/
[4]http://en.tera.gameforge.com/

Cheat detection

Current companies developing MMOGs are still afraid of going into a fully decentralized approach, as it entails a lot more risks than centralized solutions: no complete solution exists regarding the security and protection against cheating in P2P environments. And even though the client/server model is the current state of the art solution in the gaming industry, client/server models do not answer all forms of cheating attempts in MMOGs.

Even centralized, gaming servers have to disseminate information in order for the game to remain up to date for players. Some malicious players may want to use the information they receive in the game to acquire an unfair advantage against the other players. Archage [7], a centralized MMOG, had this issue publicly revealed as their servers propagate local informations to the entire connected playerbase. The game client then filtered which content had to be displayed locally. Cheaters got unfair advantages by modifying the default game client behavior to exploit this worldwide data. Official reports from game companies frequently announce "waves of bans" [8, 9] to reply to cheating attempts. Waves of bans correspond to the action taken when a set of cheaters has been detected over the past time period. These ban waves result in a denial of the access to the gaming server for these groups of cheaters. The problem behind these ban waves is the amount of data they require: every action every player makes in the game has to be recorded in order to be able to detect the cheating attempts afterwards, thus hitting the scalability of MMOGs.

A majority of P2P solutions that try to handle cheating attempts focus on reproducing trusted detection algorithms that are commonly running on centralized game servers [10, 11]. Nevertheless, none of these solutions fully address all the needs of an MMOG as many of them target small numbers of players, while MMOGs host hundreds of thousands of players [4].

Scalability of the matchmaking service

In most MMOGs, people have to get together to accomplish various tasks. To do so, a player can call friends for help, or team up with other players on the server.

In the gaming industry this service is called matchmaking: it is supposed to create a good match between avatars. For some games, matchmaking simply is not functional. Two main reasons often lead to this: lack of players or an inefficient implementation. As an example, in Smite [12] the matchmaking service suffers both from an insufficient player base and from the precision of the matchmaking algorithm [13].

In the literature matchmaking is a complex problem studied mainly in the domain of multi agent systems and aims at a perfect match over time [14–17]. The constraints are not the same in gaming: game matchmaking aims at minimizing the differences of latency between the players and the server [18] while also reducing as much as possible the response time [19, 20].

Current matchmaking approaches are already flawed and inaccurate even if clients come from a single server [12]. As the matchmaking problem is scaled up into a distributed, highly scalable environment the latter need a deep optimization of its current mechanisms and algorithms.

1.2 Contributions

My main contributions are threefold. I gathered publicly available data into an archive containing millions of user gaming sessions of an MMOG. I used this database to study flaws in a crucial MMOG service: matchmaking. My work shows that the studied game at matching both accurately and in a timely manner. I proposed a wide range of possible implementations for matchmaking services. I conducted this analysis with different player base sizes in order to depict the impact of the popularity of a game on its matchmaking.

I then designed a new way to identify trusted nodes in an MMOG network to detect cheat by gathering trustworthy nodes. I employ a reputation system inspired from the PID (proportional integrate derivative) model that will cooperatively test the nodes that are connected to the network. Based on this reputation system I tested my cheat resistant platform for MMOGs both in simulated and real environments.

I finally provided an extension of my cheat detection service for MMOGs: a fault detector for generic systems. This fault detector is compared with Bertier's and Swim's fault detectors, two state of the art approaches known for their good performance.

I tested the fault detector with an injection of random failures but also real traces from the failure trace archive (FTA) [21] that allowed us to reproduce the behavior of P2P environments. I showed in that my detector behaves better than the other studied detectors under these conditions.

1.3 List of publications

Journal Papers (2)

2015 Maxime Véron, Olivier Marin, Sébastien Monnet, Pierre Sens, “Etude des services de matchmaking dans les jeux mutlijoueurs en ligne: récupérer les traces utilisateur afin d’améliorer l’expérience de jeu”, In Technique et Science Informatiques (TSI), Hermes, pp. To be published, 2015.

2014 Maxime Veron, Olivier Marin, Sébastien Monnet, Zahia Guessoum, “Towards a scalable refereeing system for online gaming”, In Multimedia Syst., vol. 20, no. 5, pp. 579-593, 2014.

International Conference Papers (1)

2015 Olivier Marin, Sébastien Monnet, Maxime Veron, Sens Pierre, “RepFD - Using reputation systems to detect failures in large dynamic networks”, In The 44th International Conference on Parallel Processing , ICPP 2015, Beijing, China, September 1-4, 2015, 2015.

National Conference Papers (2)

2014 Maxime Véron, Olivier Marin, Sébastien Monnet, “Matchmaking dans les jeux mutlijoueurs en ligne : etudier les traces utilisateurs pour ameliorer l’experience de jeu”, In Conférence d’informatique en Parallélisme, Architecture et Système, Neuchâtel, Switzerland, 2014.

2013 Maxime Véron, Olivier Marin, Sébastien Monnet, Zahia Guessoum, “Vers un système d’arbitrage décentralisé pour les jeux en ligne”, In RenPar’21 - Rencontres francophones du Parallelisme, Grenoble, France, pp. 9 p., 2013.

Workshop Papers (2)

2014 Maxime Veron, Olivier Marin, Sébastien Monnet, “Matchmaking in multi-player on-line games: studying user traces to improve the user experience”, In Proceedings of the 24th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV 2014, Singapore, March 19-20, 2014, pp. 7, 2014.

2012 Maxime Veron, Olivier Marin, Sébastien Monnet, Zahia Guessoum, “Towards a scalable refereeing system for online gaming”, In 11th Annual Workshop on Network and Systems Support for Games, NetGames 2012, Venice, Italy, November 22-23, 2012, pp. 1-2, 2012.

1.4 Organization

This thesis is organized as follows: Chapter 2 describes the underlying problems of MMOGs architectures and covers a wide spectrum of existing solutions for the creation of a scalable cheat proof MMOG platform. A study of a crucial service for online gaming, the matchmaking, is described in Chapter 3. Chapter 4 depicts my scalable solution for cheat proof gaming. The extension of my work on the cheat proof platform, my generic fault detector, is described in Chapter 5. Finally in Chapter 6 I conclude this thesis and depicts some perspectives my work opens.

Chapter 2

Decentralized services for MMOGs

Contents

This chapter describes the current approaches for decentralized MMOGs and depict why there is still room for improvement in this domain. Section 2.1 depicts the MMOG concepts and designs. This allows a better understanding in section 2.2 of the study of matchmaking approaches, of the cheat detection issue in section 2.3. Finally, section 2.5

list the principles of reputation systems as I use them to reply to cheat detection during this thesis.

2.1 Context

The main category of MMOG is the massively multi-player online role playing game (MMORPG): the role playing genre fits well with grouped and online gameplay. Role playing games also benefit from being held on a multiplayer server as they take place in a persistent virtual world that can be shared between all players. The client-server model is the standard for MMOG design. This model is used by most current MMOGs such as Tera [1], Aion [2], minecraft [3], or World Of Warcraft [4].

Game companies servers must therefore handle a tremendous quantity of data related with avatar storage. Those environments need to be lively as they host a lot of mutable data. Big data streams continuously flow between the MMOG storage and the memory of the server. Although the power of machines grows, these huge worlds as well as data related to the players are impossible to store on a single machine.

A common approach is to duplicate the universe into separate entities. Thus, each server manages a portion of the population and only players of located in the same server can meet. A limit on the number of players per server prevents overload. In Tera [5], players can fly rapidly between universes. Overloads are avoided by allowing players to migrate only to the least loaded servers. The advantage of this strategy is that players balance the load: the normal behavior of a player is to choose the least loaded server in order to obtain the lowest latency.

As an answer to these servers scalability, many solutions [22–25] for distributed management of virtual worlds rely on P2P overlays. A P2P overlay is a topology applied to the virtual universe and entirely managed by a network of peers. This structure must not only host the world, but also handle its maintenance. This involves some locality of data in order to update or create data for players. Each type of P2P overlay has distinct properties, hence the wide variety of overlays currently in the literature. Some favor data locality, others provide more efficient routing. Whatever the P2P overlay, peers host the structure. Those P2P overlays must remain lightweight to not overburden peer bandwidth within a virtual environment which has game constraints such as:

[1] http://en.tera.gameforge.com/news/index
[2] http://www.aionfreetoplay.com/website/
[3] http://www.minecraft.net/
[4] http://battle.net/wow/
[5] http://en.tera.gameforge.com/

data persistence: Data from a virtual world must remain available and must not disappear when peers leave.

data accessibility: The stored data must be accessed by precise coordinates to allow players to move and interact with the universe.

locality management: Data locality is here to match neighbors in the network with neighboring data of the universe in order to minimize latency.

Latency is known to affect the gaming experience negatively [26, 27]. However, overlays are complex structures calculated by neighborhood graphs and must always be up to date. Updating and building graphs is expensive in terms of message complexity and calculation.

Overall there is a distinct and clear trade of between P2P overlays for MMOGs which target high scalability but require a redesign of the current architectures and client/server approaches which replicates the same small game universe on multiple servers. In both categories, many solutions exist in the literature and have shown to be sufficient for nowadays MMOGs.

Therefor in the context of this thesis, I do not focus on offering a scalable gaming universe management but target at scaling the other crucial services of MMOGs: matchmaking and cheat detection.

2.2 Matchmaking approaches for MMOGs

Solving the matchmaking problem will not benefit the gaming industry only. In [28] students are automatically matchmade into groups that fit their skills and academic requisites in order to form groups that are more suitable for each student. And [29] uses a matchmaking system to improve the e-marketplace.

Managing to match players both accurately and fast is an open issue and game developers often communicate their approaches publicly [13]. This is to reassure the community about both the efficiency and the fairness of the system. Matchmaking systems inherit from the trueskill algorithm by Microsoft research [30] as a base to evaluate player performance in the game. Besides Elo performance (described in the following section), other criteria matter just as much in a cooperative game, such as the spoken language. In [19, 31–33] latency is the parameter studied and shown as requiring a balance between matched entities, even though games rarely take latency into account for matchmaking.

In [20], the matchmaking of Ghost reckon online is deeply analyzed but this game does not rely on the same conditions as MMOGs as it targets FPS (First Person Shooter) games. More advanced solutions use agents to solve the matchmaking problem [15, 16, 28] but all of them lack a performance study in a real environment to verify its efficiency. Most of the solutions use P2P network and does not offer possibilities for a centralized server architecture [18, 29]. As the current MMOGs rely on client/server architectures, those existing solutions cannot be applied without an intensive redesign of MMOGs platforms.

2.2.1 Elo ranking

In order to introduce the gaming matchmaking concepts I need to briefly introduce the Elo ranking and its definitions.

The Elo rating system [34] created by Arpad Elo is an improved method for calculating the relative skill levels of chess players. Today many other games, including multiplayer competitions, have adapted it for their own use. Thereafter, I use ranking, Elo, or MMR (short for *MatchMaking Rating*) to talk about player ranking values.

It is assumed that a player's performance varies from game to game according to an approximately normal distribution; a person's Elo rating is the mean of that distribution. A person with a high Elo is someone who performs better on average than a player with a low Elo. This score is determined by win/loss statistics with respect to other players using a variant of the algorithm called *the algorithm of 400*. For players A and B with respective Elo ratings of Ra and Rb, the probability Ea of a victorious outcome for player A is computed as follows:

$$Ea = \frac{1}{1+10^{(Rb-Ra)/400}}$$

For each 400 points of distance between players, the player with the highest score is ten times more likely to win as the other player. This is the standard computation for Chess and it may differ in League of Legends. The actual outcome of a game is compared to the expected outcome and each team/player rating is adjusted accordingly. As a result, the score of a victorious team changes less if it was expected to win than if it was expected to lose. Successive games eventually bring each player/team to a point where they are expected to win 50% of the time against opponents of equal score.

Ea is then used to compute the new score of players following this formula:

$$Ra_{new} = Ra_{old} + K * (Sa - Ea)$$

Sa is the result of the game and is presumably 1 for a win and 0 for a loss.

Parameter K in the formula determines the magnitude of the score change. A dynamic K value is associated with each player to inhibit Elo changes after games between evenly matched opponents and to prevent inflation of the highest Elo ratings. The reason for the latter is that excessive Elo differences produce a feeling of unachievable goal for new players. In chess, the initial K value is big (25 for the 30 first games) resulting in large changes in Elo. Thus a player can rapidly find her/his correct place in the ranking system. As the number of wins and losses becomes more even, K decreases gradually ($K = 15$ to 7).

2.2.2 League of Legends case study

This section describes my study case: the matchmaking service of League of Legends (LoL). LoL is a popular game with a community regrouping over 12 million players worldwide per day in 2012 ; this ensures an abundance of user traces from a genuinely successful online game.

It is a competitive game where ranking is a central element; as such, ranking is a precious metric for studying player behaviours and extrapolating quality of service. Finally it is a session-based game with relatively short sessions, averaging around 34 minutes according to the game developers. Its matchmaking service is used often and plays an important role in the overall game experience.

League of Legends: a brief overview

League of Legends is a multiplayer online battle arena (MOBA) video game developed and published by Riot Games. LoL players are matched in two opposing sides comprising five teammates each. Both teams fight in an arena in order to destroy the enemy team's main building called *nexus*.

The LoL server definition of a *game* is the brawling session that pits players against each other inside the arena. I will therefore use this term to refer to the players' games history.

Three factors are crucial to the gaming experience: waiting times, matching accuracy, and server response times. With an average waiting time of 90 seconds in between sessions, players can spend a lot of time waiting for a session to begin (see Subsection 3.2.1). This is especially true for very skilled players: their scarcity makes it harder to match 10 such players together. Matching players with disparate skill levels reduces waiting

FIGURE 2.1: Equivalence between ranking and *Leagues*

times significantly, however it can reflect poorly on the experience. Unskilled players will feel helpless against much stronger opponents, while the latter will spend time on an unchallenging session with little or no reward to look forward to. Orthogonal to these issues, server response time is crucial in this game which requires extremely sharp reflexes. Lags caused by the servers often increase the ping by up to 300%, which severely impedes the gameplay.

LoL is a competitive game where ranking takes a significant part, both in terms of matchmaking and in terms of player status. Rank defines the skill level of a player, and rank improvement is the main goal of most LoL players. Once players are ranked, they get placed in competition categories called *Leagues*, and subcategorized into *Divisions*. This ranking, as it can evolve quickly over time, needs to be computed precisely and that is why most games use the Elo rating system. Thereafter, I use ranking, Elo, or MMR (short for *MatchMaking Rating*) to talk about player ranking values.

Figure 2.1 describes the equivalence between *Leagues* and the rank of a player. The formulas used in League of Legends for ranking calculations have not been disclosed publicly. However, most ranking implementations share the same bases inherited from the original Elo rating system. Riot Games developers do divulge interesting information about the matchmaking in LoL through its website and dedicated forums [35] , though. Based on this information and on my own data analysis, I inferred the general rules that guide LoL player matching.

Matchmaking in League of Legends

League of Legends uses a system with dynamic K values. All players start with an Elo of 1200 for their 10 first games. From there they are assigned a score, and changes occur smoothly according to the wins and losses.

League of Legends players can join several types of games, associated with different *queues* on the servers. A group of persons joining the same *queue* in order to play together in the same team is called a *premade*. A *premade* can comprise from one to five players. *Normal* games do not count towards official ranking, whereas *ranked* games do and are only open to seasoned players (above level 30). My study focuses mainly on *ranked* games since they draw together players whose statistics are more likely to be representative of the game's core community. Also, fair matchmaking is more critical for *ranked* games since they induce real stakes for the players.

According to Riot Games developers, matchmaking in LoL is based on: player ranking, the experience of each player (number of games played), and the size of the premade.

Player ranking weighs most since it has the highest impact on the outcome of the game. LoL tries to match teams as fairly as possible: it computes the average of the player ranking values for each team, and then uses these averages to match teams similarly to one-versus-one fights. This solution speeds up the matching of multiple players, yet it may result in an unbalanced game if two players with very different rankings join in a premade since the least skilled player is likely to face a highly ranked opponent. In a competitive context, this kind of quick fix is very unsatisfactory.

Adding the personal number of played games as a matching criterion alleviates this issue, but it also increases the average waiting time drastically [36](roughly around 50%). This shows the direct impact on the average waiting time of adding a single preference to the matching system.

Premades also increase the waiting time, and can even induce unbalanced battles. If the matching system cannot find an opposing team with a premade of the same size and level in the allotted time (roughly 30 seconds for a normal game), it gathers single players with higher individual ranks to build an opponent team. The higher rank is supposed to compensate for lesser coordination in the opposing team, but usually leads to a victory of the higher ranked players.

2.3 Cheating in MMOGs

In any MMOG the community of players builds a real economy in the virtual world, either in terms of ranking or in terms of virtual currency. Yet the community can have a negative influence on the game experience as players cheat in order to get unfair advantages over other players. One does not want to join a game that is full of cheaters.

There are many ways to cheat in MMOGs, which range from small, undetected advantages acquired by the player to deep modifications of the world state that completely unbalance the game economy.

2.3.1 Gold farming - use of normal behavior for illegitimate reasons

Some cheaters pay third party services to play on their accounts while they are away from their computers. During the hours the player is not connected service employees come to take his place and play for a determined amount of time.

These cheaters gather an unfair advantage quickly: any particular resource they requested when purchasing the service. This behavior known as gold farming is illegal as the player have to spend money for the service, which renders the player financially advantaged in a game where all players are supposed to have the same odds. It is called gold farming as it is the most common currency of MMORPGs.

Gold farming is very difficult to detect, given the quasi-standard behavior of employees who gather the currency for the player. If the company that provides this gold-farming service is always in the same area, a study of the rate of currency generated by the game allows identification of gold farmers.

2.3.2 Trust exploit

Another category of cheating attempts, and the easiest one to implement is the *trust exploit*. It relies on exploiting the confidence a player can have into a service while the service is actually malicious and gathering the player sensible information. These are often easy to implement because they rely on exploiting software weaknesses.

An example of such cheating is the *"map hack"* that reveals an entire map when only a part of the map is supposed to be visible to the player. These hacks are possible when weak servers [7] actually send the entire map information to the player and hide it graphically in the game. The cheater only has to block this "in-game hiding" by manually overriding a setting in the graphical driver parameters to view the entire map.

2.3.3 Bug/hacks exploit

"Bug exploit", "Hack exploit", those terms express the most common type of bugs that users will exploit in order to get advantages. Those bugs target two type of modifications: short-term and long-term.

Short-term modifications aim at modifying game content directly in the RAM. An example of such changes is FunnyPullBack [37] in LeagueOfLegends. This software developed by a third party injects its code into the game client, and returns a snapshot of the game state to a script engine. The scripting engine offers to execute commands on the game after analysis of the game state. With this tool, users can create their own scripts to automate actions in the game very accurately.

Long-term modifications address data saved on the storage medium of the client. These attacks are harder to achieve due to the protections of the program deployed by the server. But they offer durable benefits for the cheater: changes to backups of his avatar, its appearance, its objects.

When insufficiently checked by the server this category of cheats often leads to games full of cheaters. As an example, Diablo2 [38], had two multiplayer modes: Closed Battle.net, and open Battle.net. The closed mode forces to create characters on blizzard servers (and were therefore controlled by the server), the open lets users play characters stored on their hard disk and connect to one another in a peer to peer model. Once the correspondence between hexadecimal backups and the character's characteristics in game had been made, a team created the *Hero Editor*. The open Battle.net quickly became the kingdom of cheaters fighting each others.

The legitimate playerbase resorted to closed servers as it was their only option. In closed servers, avatars were deleted if inactive and players faced an added latency compared to local area network (LAN) gameplay, which reduced their game experience quality.

2.3.4 Cheating by destruction / deterioration of the network

In a game with ranking, protection against attacks that target the network is crucial. A cheater that manages to prevent opponents from joining the game can get easy victories by forfeit, climbing in the rankings.

A variant of this cheat is the creation of fake accounts to raise the level of a real player account. By influencing the system so that the server matches the legitimate account with the fake ones, a cheater mimics a sybil attack. This technique was used in League

Of Legends [6]: a player was using the community to organize games where he could easily win [39].

Another network related cheat is to delay data emissions. In FPS or fighting games where response time is important, when a player takes an abnormal amount of time to answer a technique is used to avoid artifacts of teleporting players to appear: the player avatar will follow an estimated trajectory based on the direction vector.

A well known cheat is to use this technique to make enemy players see an action in a fake direction, or froze the avatar. The cheater sends two messages: one announcing a start in the wrong direction, and the other to take advantage of the surprise generated by the abrupt change. A simpler and mechanical way to do so is to use a lagswitch [40] that will block any receiving/incoming data. If the player is hosting the game, he will still be able to play locally while everybody will see a frozen avatar. On a second press of the button, the console will then resume messages emissions and update the other players.

2.4 Cheat detection

This section present the possible countermeasures to cheat in MMOGs. Solutions can either be hardware based, or by creating control authorities to assess players actions.

2.4.1 Solutions based on hardware

Solutions based on hardware have been adopted by manufacturers of video game consoles from the beginning. Hardware components provide the means to check whether contents loaded in the console are legal or not. For example, upon insertion of a DVD into a Nintendo Wii console, the system input/output driver moves the DVD reader head in the security zone of the DVD before executing any game code section. This allows signature control and possible rejection of the inserted disk. The Nintendo 3DS goes further: it stops working indefinitely if an illegal cartridge is inserted [41].

The problem with this solution is that the protection is embedded in the hardware and becomes useless once it has been circumvented. "Chip" soldering on console motherboards re-routes the security check and always claims that the content is legitimate.

Sometimes the system console software itself includes a fault that can go undetected by hardware verification. Such was the case of the Wii, which had a breach in one of its

[6] http://euw.leagueoflegends.com/

first games. A software design flaw allowed players to get the keys used to sign content: a player managed to recover backup data. By modifying a backup and signing with the key, cheaters gained access to an area called "debug". Once the fault was found, hackers could exploit the debug area and inject a buffer overflow [42]. Sony PS3 supported the linux operating system virtualized on the first PS3 generation, which allowed access to a runlevel where a malicious user could execute unsigned code and cause a long chain of security breaches that finally ended in running unsigned content on the console [43].

A thorough analysis of all cheat detection systems describes the long list of hardware mechanisms that exist as been made by Feng, Wu-chang and Kaiser [44]:

- Study of RAM,
- Listening to system calls,
- Listening to hardware,
- Using coprocessors to check calculations etc.

Hardware based solutions alone are too vulnerable in the long term. MMOGs specific solutions using hardware are cited in the literature such as: keyboard key-press detection [45], for example, to discover bots. Bots are software used to perform actions on behalf of the user. They enter orders directly into the game client without using any input device.

StarCraft2 includes a protection-oriented listening hardware [46] that prevents automatic actions by allowing only one in-game action per keystroke.

World of Warcraft also included tests to detect bots automatically. By analyzing the keyboard / mouse activity they automatically disconnect players after a long period of inactivity.(around 30 minutes).

Hardware detectors are rarely used alone because they cannot be updated and therefore do not offer long-term security. Once a cheater exploits a flaw in the physical protection the game company cannot change all previous versions of the hardware, rendering it obsolete.

Downloadable content is strongly disliked by most players, but it is very convenient alongside hardware solutions. If the executable binary game is decrypted, the game company can apply and distribute an update that changes the encryption system. There is no backward compatibility to manage: game servers can directly refute the old key and propose a software update to any version with the old encryption system.

2.4.2 Solutions based on a control authority

Many solutions rely on the usage of a trusted authority that will assess clients requests. The following subsection describes the different types of authority.

Solutions based on a centralized authority

In the client/server model, the server is an entity controlled and monitored by the creator of the game. In order to detect simple attacks (like modifying the contents of a packet), the most trivial solution is to host the state of the game on a centralized trusted server. Thus the client does nothing more than communicating and respond to information it receives from the server. This is the case with most current MMOGs.

A protected application checks what is sent by the client. Such solutions have a cost on the server load and the amount of data that is transferred. The server may skip some cheat detection cycles to reduce the load. SpotCheck reduces the computation overhead by 50% [47].

Distributing some data to the client and delegating control of communications reduces the computational load on the server. In [48] an authority checks every transaction (the transaction is then marked valid or invalid), but the client generates the hash of the transaction. The authority only verifies that hashes are correct and correspond to a nominal behavior. One can imagine a solution coupling this approach with SpotCheck to limit the computational load of the authorities.

The problem with these control techniques is that they are cheat-proof. If a cheater passes the tests, it does not mean he did not cheat. It just means he was able to adapt to the control either using undetectable cheats or by respecting game rules like gold farming.

Peer to peer oriented solutions

In a peer to peer environment it is difficult to monitor network similarly to centralized servers. Some P2P solutions use centralized authorities to help ensure security in their network [48, 49]. The solution of Josh Goodman [48] is both a centralized solution and a peer to peer one as it relies on a centralized authority that peers of the network can assist. Peers will regularly become authorities and are able to handle requests for the server. Watchmen [49] targets smaller number of players (up to 48) by focusing on FPS type of games but covers a wide range of cheat detection.

Jardine, Jared and Zappala [50] established a hybrid architecture using both computational power offered by the P2P model and security offered by the client/server model. Hybrid solutions offer a good compromise between size and maximum network security, but the centralization of security always induces high bandwidth consumption on servers.

Peer to peer environments open more potential breaches and types of attacks as they give a part of the control to the nodes of the network. Sybil attacks aim at yielding more information than any single node should have, or at attacking a network node with multiple identities. There is also the possibility of nodes that collude in the peer network. The first defense is quite direct and explicit: the goal is to verify that peer "A" is not malicious. The state of A is sent to peers, who then calculate the same actions as "A". After calculation, they can check that A returns the correct result. These checks can be made either:

- by testing with a softcoded algorithm in the application: these tests are vulnerable without regular updates.
- by tests generated randomly at runtime of the application: testing is then more costly to maintain but are more reliable over time [51].

Rules defined cheat detection

This section lists some of the approaches based on the existence of rules in the universe that allow the cheat detection mechanism to detect incorrect behaviors.

Mathematical properties of a game can prevent cheating. Pittman [52] proposes a solution for card games where two peers can assess what actions are possible or not. A state machine describing the state of the game determines which actions are possible at any given time. Such solutions offer opportunities outside the server. State machines and mathematical laws establishing the behavior of players in MMOGs are far too complex and have yet to be written.

Bots can be detected relatively easily because it has been shown that bots do not have the same behavior as humans and their movement traces are fundamentally different. Agent based solutions allow effective bots detection [53, 54]. These small entities provide a good local analysis of a simple problem at a low cost, and work together when the problem needs to be solved in a distributed manner.

To protect against attacks exploiting data flow, mobile guards [55] encrypts memory. Guards are the only ones able to read the contents of the game distributed at the launch.

Scattering data on nodes in the network is an alternate possibility. The major drawback is that removal of data becomes very slow.

Delap and Al [56] run the client through a checking application to assert that it is indeed the original code which was launched. This can prevent attacks targeting long term storage modification.

Finally to protect against typical attacks that use duplication of identity, a solution is to make the second authentication more difficult than the first. With crypto- puzzle [57], clients are required to perform complex and very costly computational tasks. Acquiring many identities cost a lot of time and resources, thus reducing the appeal of Sybil attacks.

2.5 Reputation systems and their mechanics

This thesis plans to detect cheat in MMOGs using the concept of reputation to describe the behavior of a player in the game. A player with bad reputation would be ultimately seen as a cheater.

A reputation system is a distributed algorithm for obtaining global feedback on every node of the network. For example in CORPS [58], reputation feedbacks are used to establish a trusted network based on nodes from the P2P system. This trusted network is then used to route packets efficiently, and enhance the global transfer rate of the network. The number of hops and the number of re-emissions decrease when using a network built with trusted nodes.

Reputation systems are not only used for IT algorithms and software. For instance, an auction website can use a reputation system in order to judge its users. Gathering feedback on eBay helps building a reputation scale between buyers/sellers.

A reputation system is not fully reliable, as malicious nodes can also send feedback. [59] gives multiple definitions of trust, depending on the type of relation between peers of the network :

- dependency trust: a peer expects another peer to behave in a certain manner based on the *trust ratio*. A trust ratio is an expression of a confidence percentage regarding another node of the network. The higher the trust ratio, the more likely the other node is correct.

- decision trust: peers rely on the actions of others peers to take their decision, even if some of the actions were malicious.

Reputation is highly related with trust, but their meaning differs. Reputation is what is said or thought about another entity and its actions. Because of this slight difference, a node can say of another one:

- I trust you because of your reputation
- I trust you even though you have a low reputation

Those two sentences show the difference between a reputation value and trust. Trust is a subjective view created from past experiences between two entities, while reputation is more like a global view gathering subjective views in a collective way.

2.5.1 Effectiveness of a reputation model

In [60] Yang Rahbar provides a deep analysis of the six important characteristics of any reputation system:

- High precision : to help distinguish correct peers from malicious peers, the reputation system must calculate its opinions as accurately as possible.
- Fast convergence speed : the reputation of a peer varies over time, the speed of convergence should be as fast as possible to allow to reflect these fast paced changes.
- Low overhead : the system must consume a small portion of available resources such as memory, CPU or bandwidth.
- Adaptability to peer dynamics: the system is a distributed environment in which there is a high rate of peer connection / disconnection, called churn. The reputation system must therefore be resistant to this kind of behavior and should not spend its time rolling back to the initialization phase.
- Robustness against malicious peers: the system must be resistant to attacks from malicious peers as well as from grouped attacks.
- Scaling : the system must scale both in convergence speed, accuracy, and overhead for each peer.

2.5.2 Reputation system designs

A reputation system often works in 3 steps :

- Initialization: start of the network.
- Convergence: slow convergence to a system where all nodes are evaluated.
- Update: all nodes are already judged but their states are updated.periodically

A reputation system can be seen like the feedback system of eBay [61]. Its handling can be done in a centralized manner with a server, which will then compute and gather feedback from all peers. But it can also be fully decentralized and therefore the peers need to aggregate feedbacks [60]. The problem of letting peers test themselves goes back to the cheat detection problems in P2P MMOGs: one cannot know for sure that a peer feedback is being honest.

Reputation systems must have the following three properties [62]:

1. entities should have a long life, so that after every first interaction there is always a possibility of future interaction.
2. assessments on current transactions are recorded and distributed.
3. assessments regarding transactions should influence and guide decisions on transactions.

It is crucial to encourage nodes to participate. A reputation system needs regular updates. Without these evaluations it is impossible to assess the reputation. One way to encourage peers to participate is to provide incentives. In PlayerRating [63] peers that do not participate in the system cannot access the ratings.

This principle is often used in websites moderated by users like SlashDot [7]. Peers receive regular opportunities to moderate comments. Moderators with positive reviews are more often granted the right to moderate.

The computation of peer reviews follows different models ranging from a simple average of the opinions received to the use of a Bayesian rule. Mixed solutions as in WTR [64] include an approach where nodes monitor areas and proxy traffic in those zones. These super entities can thus control the behavior of nodes under their responsibility and evaluate reputations.

[7] http://slashdot.org/

Reputation values are a major part of the reputation systems. In order to generate them it is possible to consider either direct or indirect interactions.

2.5.2.1 Direct Interactions

Direct interactions base themselves on the association of a reputation value modification per transaction. There are three major models for direct reputation assessment:

Average ratings

Average ratings are used for instance in e-commerce services such as Ebay and Amazon. EBay positive reviews are counted as +1 positive or +1 negative reviews. There is no upper bound on the counters, peers can have very high marks. This type of model follows a "power law" described in [60]: peers with high marks will become super peers and peer isolation (peers without reviews) will grow in the network. Figure 2.2 shows a powerlaw distribution according to this model. Peers with high marks are very rare while the majority of them will remain non evaluated. The Amazon model is based on averaged notes. Although analogous to eBay where sellers with good grades will be promoted, this model limits the gap with new young peers.

Bayesian models

Bayesian systems take binary values as notation inputs (positive or negative) and then use beta probability density functions (BetaPDF) to compute feedbacks. The result of these functions combines the old rating opinion with the new. BetaPDF expresses the view of a reputation as a tuple (α, β) representing the amount of positive and negative reviews. These functions provide a basis for calculating changes in reputation. Youtube videos possess a number of "Likes" and "Dislikes" per video, which corresponds to α and β in the Bayesian model.

Reputation of discrete systems

Some systems [65, 66] use discrete values such as "Good, Bad , Average ... ". The advantage of these systems is that they are easy to understand for humans. But they make it difficult to apply the complex computation of Bayesian models. These systems are not used directly, but are often a user-friendly overlay for numeric notations. These systems are a kind of abstraction layer for reputation systems.

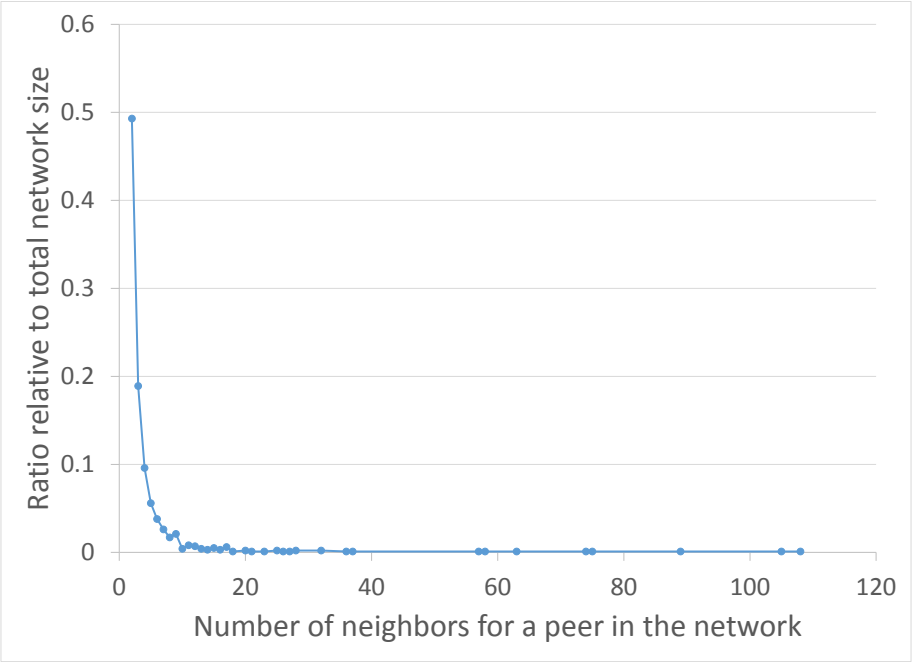


FIGURE 2.2: PowerLaw distribution generated with NetworkX [2] python module

2.5.2.2 Indirect feedback

Indirect judgments induce transitivity of reputation values. Transitivity of trust accelerate convergence but opens vulnerabilities.

If a Node B carries out a good transaction with node A, but node B trusts four nodes wishing to downvote A. The nodes wishing to down vote A only need to send negative reviews to B to sway its opinion about A.

Solutions to this issue include:

Involving a large number of peers

The more peers are involved, the faster the reputation will be built. There is a risk when including reputations received by other nodes messages: malicious nodes can try to falsify the reputation data they are supposed to be relaying.

To mitigate the impact of some malicious nodes we can consider that, correct nodes are in a vast majority and a simple mean will leverage the impact of incorrect nodes.

In case of systems where one cannot assume the correctness of nodes two other possibilities remain:

To counter false reviews generation by malicious nodes a good way is to increase the cost of generating a single review. Correct nodes which generate reviews sporadically are not impacted by a periodic computational cost. Malicious nodes that targets at altering multiple reputation values at high rate will have a huge summed up computational cost, rendering the task impossible.

Reputation values can be tagged with their transaction id. Therefor a node receiving a reputation message can check if reputation values mismatch for a specific transaction. The reputation computation is then computed locally on each node after aggregation of all transactions. The downside of this approach is the increased number of messages.

2.5.3 Countermeasures to reputation systems attacks

Reputation systems are often target to the same types of attacks as P2P MMOGs:

Single attacks

Isolated attacks either:

- lie about transactions to give a false impression about another peer
- oscillate between correct and malicious behavior to deceive other peers.

A peer can make a series of correct transactions in order to achieve a strong note, and then do an evil deed regularly that will not decrease the reputation too much. Solutions exist to protect against this kind of attack [67, 68]. In order to isolate the nodes who emit unfair reputations, it is possible to increase the punishment to compensate the good actions they did. A punishment corresponds to an unconditional lowering of the reputation values, thus ensuring incorrect nodes are visible as quickly as possible.

Another option protecting against single attacks is to analyze the history of actions in order to detect patterns of cyclical behavior [69]. As an example, in [70] the authors use pattern matching in order to detect intrusion into a system. In a reputation system, once a cyclic behavior is detected, the concerned node can be excluded from the network or ignored.

Identity theft

If a node manages to steal the identify of a trusted node, it can handle the requests on its behalf. This mode of attack is very difficult to prevent as it requires a system that can "authenticate" identifiers [71].

To protect against identity theft, many of the solutions requires human interaction and are therefor beyond the scope of this thesis [72, 73].

Grouped attacks

This type of attacks uses a large group of colluding nodes in the system to artificially raise/decrease the rating of a node, or oppositely bring down the rating of another node. These attacks are countered by example in eBay by associating a unique sale to a feedback. As eBay takes a commission on the sale, a "false opinion" has a financial cost thus discouraging naturally to forge many fake reviews.

Sybil attacks

Sybil attacks aim to create multiple identities to control a part of the network, and thus generate sufficient weight to influence reputation. The parry to the creation of new accounts is to judge new nodes as malicious. The system then becomes resistant to this type of operation, but converges slower. Another solution is to make it computationally difficult to obtain multiple identities [74]. This solution is expensive in terms of CPU usage but does not force the reputation system to converge more slowly.

2.6 The current state of MMOGs services

This chapter described the flaws and nature of two MMOG services. In current games matchmaking services, solutions rely on client/server approaches and do not scale well. P2P solutions exist but would require a redesign of the current MMOGs platforms. Moreover, gaming matchmaking is particular in a way that it does not always require a perfect match and will prioritize a faster answer. Existing approaches can therefor not be applied to gaming matchmaking.

MMOGs are very popular systems aimed at scaling with a high number of concurrent players. Contrary to what they were designed for, it is observable that the services need improvement not only in scalability but also in cheat protection. P2P solutions

handling virtual worlds lack a good cheat detection mechanism but scale well, while centralized ones fail at achieving the amount of concurrent players required but provide cheat detection.

I decide for this thesis to use a reputation system to assess the cheat detection part. As reputation systems have been vastly studied a lot of inherent vulnerabilities are covered, making reputation systems a good candidate for a P2P scalable cheat detection for MMOGs.

Chapter 3

Towards the improvement of matchmaking for MMOGs

Contents

The *matchmaking* for Multiplayer Online Games (MOGs), is a crucial service that allows players to find opponents. Current solutions raise important issues regarding player

experience. They often lead to mismatches where strong players face weak ones, a situation which satisfies none of the players involved. Furthermore, response times can be very long: ranging up to hours of waiting until a game session can begin.

Most game production teams focus on improving graphics and playability to compete with one another. As a consequence, game supporting software such as engines and middleware services consists mainly of legacy components that are widely reused and often quite old.

The improvement of game services requires an extensive study of how they behave when used by real players. To do so, it is crucial to obtain and analyze data from real platforms. Such data is very hard to get by because game developers want to prevent its reuse by their competitors and by potential cheaters.

In this work, I focus on acquiring real game data in order to improve matchmaking services. I use data gathered from a popular online game server [1] to show that mismatches and response times are indeed critical issues for matchmaking. I then study and compare matchmaking approaches, both theoretical ones and implementations from the industry. The main contributions are the following:

1. a description of my freely available dataset which covers information about over 28 million game sessions [75].
2. a detailed analysis of the acquired data and highlight the main issues raised by one of the game's crucial services: namely its matchmaking.
3. a study of different matchmaking approaches regarding their accuracy and scalability

Part of this work was published in NOSSDAV 2014 [76].

The chapter is organized as follows. Section 3.1 presents my dataset and describes how I retrieved user traces. Section 3.2 analyzes these traces to highlight the issues raised by matchmaking for MMOGs. I will finally define $MyMatch$, a formula of my own evaluating if a matching is valid, to compare all the wide range of different approaches I introduce in 3.3.2, and analyze their scalability and correctness in section 3.5.

[1] http://euw.leagueoflegends.com/

3.1 Gathering data about League of Legends

In order to acquire data for my study, I gathered public information from a League of Legends server. The resulting data set covers information about more than 28 million game sessions obtained over a month of crawling, and is freely available as a database [75]. This section describes the nature of the data I acquired and my retrieval method.

3.1.1 The nature of the retrieved data

I distinguish three categories of data among the dataset I acquired:

Avatar information: The first category, regroups data that characterizes a player's status inside the game. This category is common to many games such as World of Warcraft or Diablo 3, where the main purpose is avatar evolution and competitiveness. For example, the items that an avatar possesses or the damages dealt to others are fields that can be found in every role playing game and in every first-person shooter. This category is by far the largest, accounting for 43 data fields out of 77.

Company handlers: The second category, gathers content that is specific to League of Legends. The company handlers are variables that exist only to help the company maintain the game, such as player identification numbers and timestamps. This 13 data fields of this category helped sorting my players in the database. Although it represents data specific to League of Legends, some of it allows analysis for business related concerns. Such is the case with data about *skins* that players apply to their avatars. *Skins* are add-ons that players buy in exchange for real money in LoL. Relating the *summoner* level of the players in the game with the first time they played with a skin provides insight about the moment players tend to buy their first skins, and which kind of skin they first buy.

Data related to the matchmaking of the game: The third and last category intersects with the previous categories: it includes avatar information and precise network latency monitoring. The *userServerPing* reveals the average latency incurred by a player during the game. The *timeInQueue* data field is also crucial for my analysis as it represents the waiting time before a player gets matched with other players. I also use some avatar data in my LoL matchmaking analysis, such as the number of kills dealt by the avatar (*kill*), and the number of times it died (*num_death*), in order to detect imbalanced games.

3.1.2 Services used to retrieve data

To retrieve LoL information I used a free open source code [77] to pack/unpack the contents of the messages sent to the game server.

LoL servers follow a *function call* paradigm to handle requests. A message must be constructed in such a way that it identifies the right *function* from the right *service* and contains the function parameters.

An example of a typical request is : <"summonerService" "getSummonerByName" "darkKnight67">.

My first task was to examine all LoL services (there are over a hundred) to identify the ones associated with game statistics. Among the latter, I identified two specific services: the first handles the recent games history of players, and the second provides statistics about any given game once its identifier is known.

More specifically in terms of LoL server requests, I used the *getSummonerByName* function of the *summonerService* to translate *accountId* values into what RiotGames stores internally as *summonerId* values. Once this *summonerId* is obtained I can then call *getRecentGames* on the *playerStatsService* to get an array of *games* from the LoL server.

Each player history contains 10 games or less, depending on the number of games played during the last seven days.

LoL servers process game data as a hashMap of (key, value) couples, but send it through the network as raw content. A big part of my work was to redefine a proper Java class describing game statistics, with which I could grab all the fields sent by the server when requesting for games histories. Once this was achieved I fed all possible *summonerId* values to the server to obtain games histories data, and stored it into a database for long-term usage. This allowed efficient searches and statistical computations for my analysis presented in section 3.2.

After thorough verification, I discovered that the LoL servers I queried fail to fill in some data fields and hand out invalid NULL values instead. Unfortunately, such is the case for data that I deemed really valuable for my analysis: in particular the predicted win percent, and the *KCoefficient* that is used to compute the Elo gain.

3.2 Analysis of a matchmaking system

This section presents my analysis of the data concerning the LoL matchmaking service. I evaluate the impact of this service on the game experience in terms of waiting time, matching precision, and server response time.

3.2.1 Influence of ranking on waiting times

My first assessment concerns the distribution of LoL players in terms of ranking, displayed as the solid line in Figure 3.1. Besides allowing correlations between player skills and quality of service, such information is very valuable for designing and evaluating new matchmaking solutions. The initial ranking value for LoL beginners is 1200, which explains why there is a large majority of players around and just above this value. As expected the number of players diminishes as the ranking increases, since it requires increasing skills to attain higher rankings. An obvious consequence of this distribution is that matching highly ranked players together is harder, and thus should take more time.

Figure 3.1 illustrates this issue through the dashed line which represents the average waiting time with respect to player ranking. Above rank 1700 the average waiting time increases exponentially. It actually reaches up to 45 minutes for the highest ranks, but I did not include this in the graph to preserve its readability, as most waiting times are below 100 seconds.

In the LoL dedicated forums, Riot Games developers state that they aim for an average waiting time that will not exceed 30 seconds. My observations show that the slowdown incurred by the vast majority of players brings the overall average waiting time closer to 90 seconds. One might consider 90 seconds an acceptable duration; however further calculations show that, out of 825000 ranked game requests, 65259 took more than 5 minutes to get matched, affecting all player ranks alike. This amounts to 7.9% of matches that fail to start within an acceptable time-frame.

An important conclusion to be drawn from these observations is that the current LoL matchmaking system does not scale well. Both curves top out simultaneously around the Elo value of 1200. This demonstrates the bottleneck effect of the matchmaking service trying to deal with huge numbers of similarly skilled players. Since my information covers more than a month of crawling, this conclusion is unlikely to result from temporary server issues.

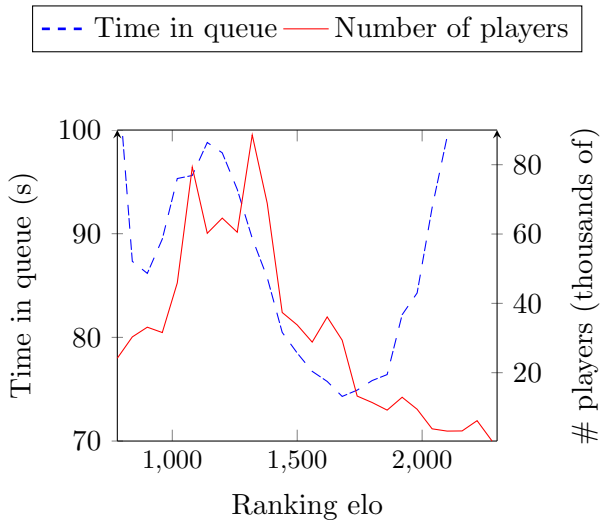


FIGURE 3.1: Distribution of the players ranks and their waiting time to get matched

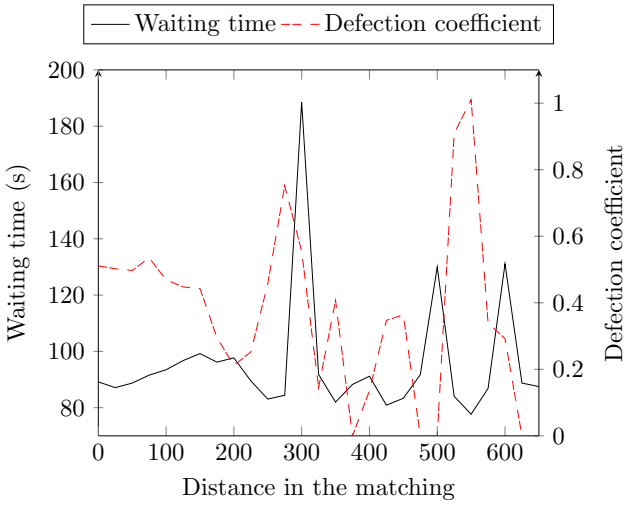


FIGURE 3.2: Matching distance and frequency of game defections

3.2.2 Impact of the matching distance on the game experience

My next assessment studies the impact of the matching distance on the gaming experience. The matching distance is computed by taking the distance between the average ranking of players of the same team and the one of the enemy team. This value is close to what Riot Games uses for its game.

Getting matched against opponents with extremely different skill levels usually results in a boring session: an outclassed player will experience a half-hour of severe pounding from the opponents, while a very superior player will have a very unchallenging session. Even though a reasonable amount of challenge in games is good, getting matched with people from two leagues away constitutes in any case a true handicap. This sometimes

results in a player simply quitting the game. It is important to keep in mind, however, that defections are rare in LoL as they often lead to banishment from the game.

For the purpose of my study, I first computed a coefficient associated with the frequency of game defections; a value of 1 corresponds to the highest number of defections encountered for any given matching distance. Figure 3.2 correlates these results with the average waiting time. Both curves are inversely proportional, which means that a quick matching resulting in a significant ranking difference often leads to a defection. The two defection peaks observed before 300 and before 600 correspond to the ranking distances between different *Ranked Leagues* in LoL. In other words, players pitted against opponents in a different category of competition are more likely to defect.

3.2.3 Impact of latency on the game experience

Even though Sheldon et al. [31] show that latency bears little importance in real time strategy games such as Warcraft 3, such might not be the case for MOBAs. Hence I evaluated the impact of latency on LoL gameplay.

To begin with, I studied the distribution of the latencies within the player population; Figure 3.3 presents my results. It shows that the ping remains between 30ms and 50ms for a vast majority of users. Since this is a very low value, my first intuition was that the results from [31] stand true for LoL. However on closer inspection, I observed that pings above 100ms were found in more than 1.2 million games, representing 7% of the total.

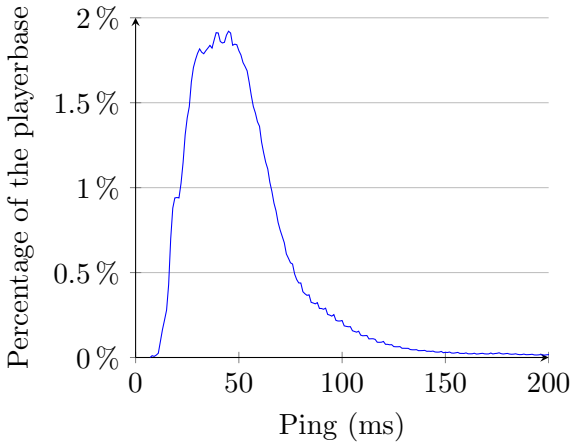


FIGURE 3.3: Distribution of the players' ping in League of Legends

Ranking is not a good metric for evaluating the effect of latency on the gameplay, as lags will impede poorly skilled and highly skilled players in similar ways. Besides it affects single players instead of teams. Therefore, I used another metric associated with

player performance: the *killed death assist ratio*, or *KDA*. It is possible to die, kill or assist in killing enemy avatars multiple times in a LoL session. Hence, the ratio between killing/assisting and dying gives a good insight on the in-session performance of a player. The formula used to compute the KDA is the following :

$$kda = \frac{assists+kills}{max(deaths,1)}$$

The KDA is computed by summing the number of assists done in game with the number of kills, then dividing by the number of death. When the player never died the KDA is a sum of the number of kills and assists. A KDA inferior to 1 reflects poorly on a player's general skills. A KDA of 0 could show a person that is disconnected for all the game duration, or making everything possible to avoid being helpful. A high KDA shows a good understanding of the game mechanics.

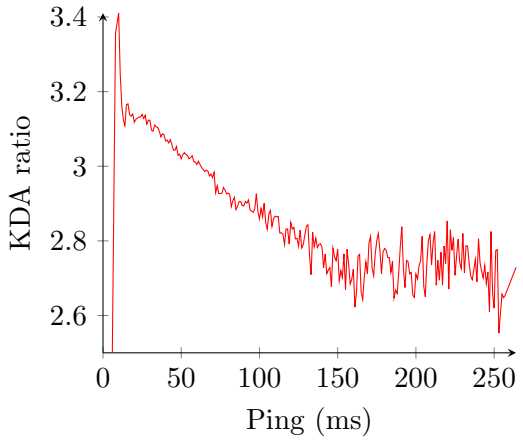


FIGURE 3.4: Impact of the latency on the player performance

The dotted curve in Figure 3.4 illustrates the relation between the KDA and the ping. Since the KDA decreases steadily as the ping increases, ultimately dividing the maximum KDA by a factor of 8, I conclude that latency does indeed impede on playability.

Latency may also affect player defections. 16% of players that leave games early incur a latency superior to 100ms. Considering that the proportion of all players with a latency superior to 100ms is 7%, high latency and premature leaves seem to correlate.

My conclusion is that matchmaking ought to include ping values as a criterion in order to improve the overall game experience for MOBAs. The same conclusion is drawn in [78]: MOBA games are close to FPS games in this aspect that latency strongly impacts gameplay. However, LoL's current matchmaking service fails to answer on time without any extra criteria for a very large majority of ranked players.

Overall, the experiments described in this section corroborate that current matchmaking architectures could use the following improvements: reduced waiting times, better gaming experience, and extended matching criteria in order to satisfy players further.

3.2.4 Crosscheck with data from another game

Defense of the Ancients 2 (DotA2), a MOBA developed by Valve, is a direct competitor of LoL. DotA2 is more recent than LoL and offers some noticeable improvements. First of all, it includes player behavior in the matchmaking criteria. A special queue called the *low priority* punishes players who misbehave: those who frequently leave online game sessions before the end, and those who often get bad reports from other players. Low priority players cannot use in-game communication and thus cannot abuse their opponents verbally. They also wait longer before getting matched, which often results in game sessions that involve low priority players ony.

Valve matchmaking includes several visible preferences that the player can select from, for example game mode, language spoken, or server location. However these preferences will be discarded as soon as the matching time increases beyond a default duration.

I gathered valuable data about the matchmaking of DotA2 from an open access website that exports the ValveAPI. This data covers two weeks of DotA2 matchmaking: figure 3.5 plots the number of matchmaking requests sent to the service as time passes, and figure 3.6 shows the average waiting times during the same period. Peaks of matchmaking requests match peaks of waiting time: similarly to LoL, it appears that the waiting time and the matchmaking requests load correlate. This comforts the conclusion that scalability is an issue with respect to matchmaking.

3.3 Tools for a better player matching

My trace study of matchmaking systems supports the necessity for improvements. The question remains which algorithms are best suited to match players together. As a first step towards an answer, this section defines a metric to measure the quality of a matching and describes six different matching algorithms.

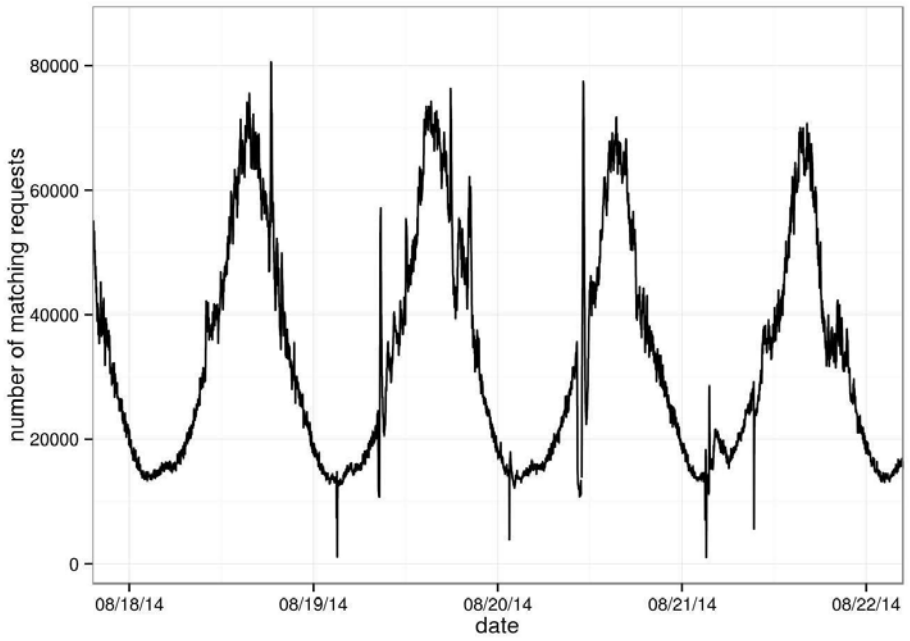


FIGURE 3.5: Number of matching requests over 4 days in DotA2

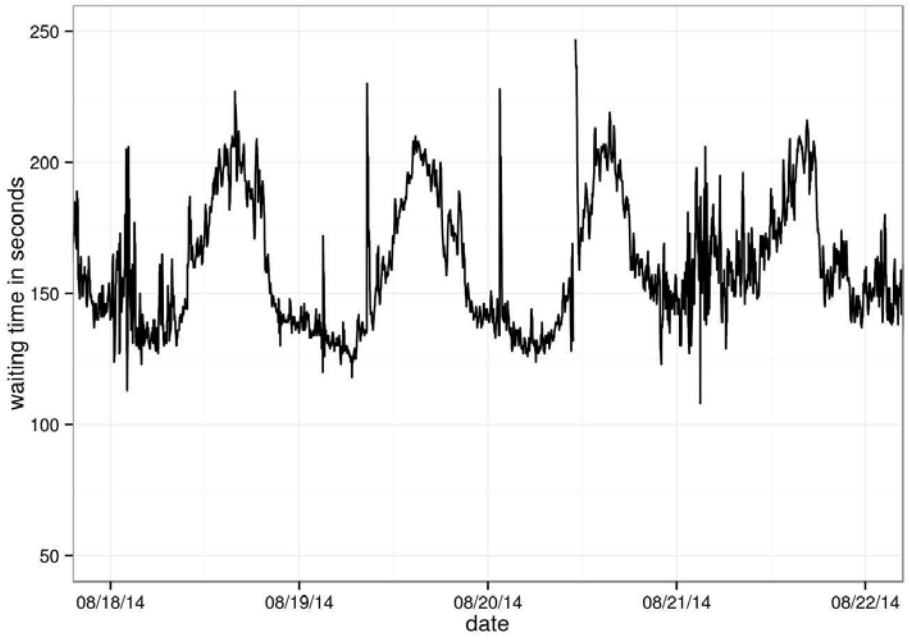


FIGURE 3.6: Average player waiting time over the same period in DotA2

3.3.1 Measuring the quality of a matching

The objective of a matching algorithm is to bring together players with similar characteristics and preferences in a minimal amount of time. A utility function is a good tool for performing such a task.

In order to propose a utility function for matchmaking, I first define the matching distance $D(p_1, p_2)$ between two players p_1 and p_2 as their weighted N-dimensional euclidean distance, with N the number of matching criteria.

$$D(p_1, p_2) = \sqrt{\sum_{i=1}^{N} w_i * (p_1(i) - p_2(i))^2}$$

Weight values w_i depend on the priorities of the game designers. To give an example: in my experiments with the LoL dataset, I match players according to their ranking and their latency. I consider that ranking is as important as latency, and I set $w_r = 0.5$ and $w_l = 0.5$ as the respective weights for ranking and latency.

Let $\delta t(p)$ be the waiting time (in seconds) of player p in the matchmaking queue. I propose the following function for matchmaking where the longer the waiting time is, the looser the matchmaking precision becomes:

$$MyMatch(p) = \begin{cases} 1 & \text{if } \exists o, min(\delta t(p) * \alpha_t, MAX_t) > D(p, o) \\ 0 & \text{if } \delta t(p) * \alpha_t > MAX_t \end{cases}$$

p is the player who requests a match, and o is a potential opponent in the waiting queue. α_t and MAX_t are parameters of the system: α_t is a weight associated with the waiting time, and MAX_t is a constant that prevents a request from lingering indefinitely in the waiting queue. This addresses cases where players have rare characteristics and are therefore hard to match.

In the case of League of Legends, I set $MAX_t = 300$ as it corresponds to the distance of three rankings. The original matchmaking scheme of LoL uses also this value. Hence setting $\alpha_t = 10$ imposes a 30 second timeout for players that prove impossible to match in a balanced game. They will afterwards be matched in a best effort manner.

Matching two groups of more than one player is achieved by computing the distance between groups as the average of all the distances between players of each group; the same average applies to waiting times. The algorithms studied in Subsection 3.3.2 will all target grouped matchmaking of five versus five players as League of Legends does.

3.3.2 Various algorithms for matchmaking

I propose six different matchmaking approaches, sorted by their increasing degree of complexity.

- *First Come First Served (FCFS) algorithm.* FCFS matches players as soon as it receives ten matching requests. It forms two teams regardless of the players' characteristics and the game can start.

- *Simple List (SL) algorithm.* SL (Figure 3.8) targets small playerbases: it maintains a single list of player requests on a centralized server, and applies function $MyMatch$ to form opposing teams.

- *World of Warcraft (WoW) algorithm.* World of Warcraft, developed by Blizzard, is not in the same game category as LoL: it is an MMORPG. However it is among the most popular games nowadays, which means that it handles a huge playerbase and must contend with scalability issues. It also proposes a game mode that is very close to a MOBA: Dungeon Finder picks players from all servers of a common virtual realm to form 5-player teams that explore a closed dungeon. Players can choose a dungeon and ask for a specific role in the team. Blizzard published the matchmaking algorithm of Dungeon Finder in a public release: WoW (Figure 3.9) picks players from a single list to form teams, but breaks incomplete teams to the benefit of larger incomplete teams. Consider two incomplete teams A and B: if $card(B) < card(A)$ then WoW removes enough players from B to complete A. WoW uses its own matching formula $WoWMatch$, which enforces a maximum ranking distance of 100 among players of a group and and waiting times under 30 seconds.

- *CutLists (CL) algorithm.* CL (Figure 3.10) is an optimization of SL: it cuts the single list into sublists and inserts player requests directly in their assigned sublist. Every sublist corresponds to a distinct range of values for a matching criteria. For example with LoL, each sublist contains players whose ranking R returns the same value for the euclidean division of R by 100; and each of these sublists gets divided into lists of players whose latency L returns the same value for $L/10$. Players in the same sublist are very likely good matches, so ranged sublists ought to speed up the matching process.

- *Multi-Threaded CutLists (TCL) algorithm.* TCL (Figure 3.11) is a further optimization of CL that exploits concurrency. The server creates as many isolated threads as there are sublists for the first matching criterion, and a coordinator thread (Figure 3.11) dispatches the matching requests.

- *Peer-to-peer (P2P) algorithm.* The last approach is a fully decentralized algorithm on top of an unstructured peer-to-peer overlay (Figure 3.12). It introduces a two phase matching. The first phase propagates requests to close neighbors: since the distribution of player characteristics is generally very skewed, this phase will satisfy

a vast majority of matching requests. The second phase handles requests that remain after an unsuccessful first phase and targets players with rare characteristics that are hard to match: it performs a full fanout gossip to reach every node in the overlay. The list of requests allows to filter request a node already received. It is set to a tenth of the network size in order to adapt to different scale, and is at least of 100 requests.

All algorithms use a task of group management. This task described in Figure 3.7 is ran in the background and browse the list of waiting groups in order to try to match complete groups together. Once the groups have been found, the task send a reply so that the players can start the game.

Data: list of waiting groups G

```
1  %search for compatible preformed groups
2  for (Groups g in G) do
3      for (Groups g2 in G) do
4      if (g.size()==5 && g2.size()==5 && MyMatch(g,g2))
5          sendReplies(g,g2)
6          G.remove(g,g2)
7      end
8  end
```

FIGURE 3.7: Preformed groups handler task

Data: list of waiting players L
list of waiting groups G

```
1  Task 1: Upon reception of a request
2  L.add(request)
3
4  Task 2:
5  %search for preformed groups
6  for (Request r in L) do
7      for (Group g in G) do
8          if (MyMatch(r,g) && g.size()<5)
9              g.add(r);
10             L.remove(r);
11         end
12     end
13 end
14
15 %create new groups
16 for (Request r in L) do
17         for (Request r2 in L) do
18         if (MyMatch(r,r2))
19             G.add(new Group(r,r2));
20             L.remove(r,r2);
21             break to Task 2;
22         end
23     end
24 end
```

FIGURE 3.8: Simple List algorithm

Data: list of waiting groups LG

```
Task 1: Upon reception of a request
LG.add(new Group(request.player))

Task 2:
%group creation algorithm of WoW
for (Group g in LG) do
        for (Group g2 in LG) do
                for (Player p in g) do
                        if (WoWMatch(p,g2)
                        and g2.nb_Players>g.nb_Players
                        and g2.waitingTime>g.waitingTime)
                                g2.add(p)
                                g.remove(p)
                        end
                end
        end
end
```

FIGURE 3.9: World of Warcraft algorithm

Data: N times P lists of waiting nodes, NP
list of waiting groups G

```
Task 1: Upon reception of a request
x=floor(request.player.ranking/100);
y=floor(request.player.ping/10);
NP[x][y].insert(request);

Task 2:
for (i in (0..NP.xDim)) do
    for (j in (0..NP.yDim)) do
        %search for preformed groups
        for (Requests r in NP[i][j]) do
            for (Groups g in G) do
                if (MyMatch(r,g) && g.size()<5)
                    g.add(r);
                    NP[i][j].remove(r);
                end
            end
        end
        %create new groups
        for (Requests r in NP[i][j]) do
            for (Requests r2 in NP[i][j]) do
                G.add(new Group(r,r2));
                NP[i][j].remove(r,r2);
                break to Task 2;
            end
        end
    end
end
```

FIGURE 3.10: 2-dimension CutLists algorithm for LoL

3.4 Measuring up to an optimal match

In order to assess the capacity of my matchmaking formula to form balanced groups of players, I compared it with an optimal match.

I acheived an optimal match for a limited subset of the LoL traces with the CPLEX

Data: Ts: list of working threads

```
1  Task 1: Upon reception of a request
2  x=floor(request.player.ranking/100);
3  Ts[x].send(request);
4
5  Thread Task:
6  Cutlist Task 1 and 2
```

FIGURE 3.11: Multi-Threaded CutLists coordinator thread

Data: list of recently seen requests L
matched=false

```
1  Task 1: Upon reception of a request
2  if (request is a request of accepted match)
3      %player is matched
4      matched=true;
5  if (MyMatch(myRequest,request) and !matched)
6      sendMatchingAcceptedRequest(request.sender);
7      matched=true;
8  else if (!L.contains(request))
9      sendToProportionOfNeighbors(r,30%);
10     if (L.size()> max(network.size()/10,100))
11         L.pop();
12     end
13     L.insertLast(r);
14 end
15
16 Task 2:
17 %Phase 1
18 sendRequestToNeighborsTTL(myRequest,3); % 3 hops
19 sleep 3 seconds
20 %Phase 2
21 if (!matched)
22     sendRequestToNeighborsTTL(myRequest,-1); %unlimited hops
23 end
```

FIGURE 3.12: 2-phase dissemination of the P2P algorithm

solver [79]. To do so, I took the first 50 traces from the League of Legends database as matching requests and looked for the optimal matching distance with CPLEX. CPLEX set aside two of the requests as unmatchable, so I removed these. CPLEX took 2 days to compute an optimal match with the remaining 48 requests.

I then made a mathematical model of my matchmaking function, and applied it to the 48 requests optimally matched by CPLEX. To push the comparison further, I also included matches computed by *FCFS* and *WoW* for the same subset. I selected these two in particular because they don't use *MyMatch*.

Figure 3.13 gives the average matching distance resulting from each algorithm. Both *MyMatch* and *WoWMatch* produce matches that are not too far from the optimum, which is reassuring, while *FCFS* unsurprisingly produces very loose matches. WoW-Match has a better matching precision than MyMatch, but Subsection 3.5.1 shows that

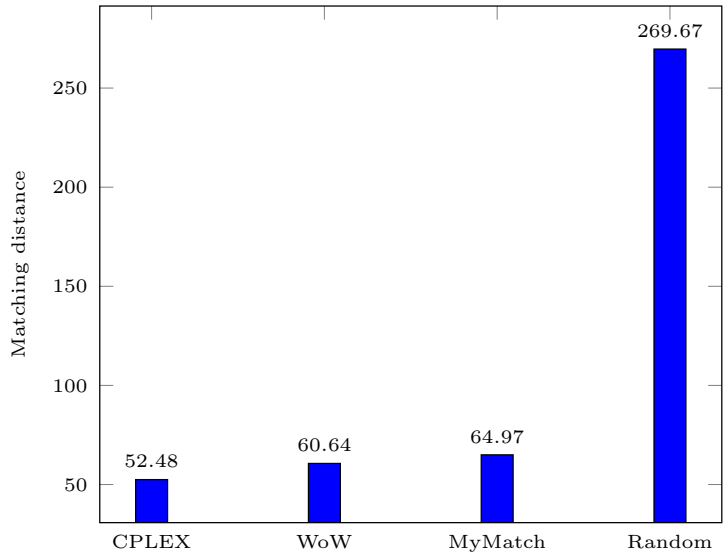


FIGURE 3.13: Comparison of matching distances with an optimal match computed by CPLEX

the price of this higher precision comes at the price of an inability to handle high rates of matching requests. Conversely, FCFS forms groups very fast at the cost of very imprecise matches; note that the matching precision of FCFS remains within the limit of 300 officially enforced by LoL.

Now that I've shown my matching formula to produce acceptable matches in comparison with an optimum, I follow up with a performance comparison of matchmaking algorithms along several metrics described hereafter.

3.5 Performance evaluation

The aim of the present performance evaluation is to study the capacity of each algorithm in terms of matching. To remove the influence of the network on my results, a single client process runs on the same machine as the server, requests are sent in UDP to get the maximum bandwidth, and UDP sockets are set with a high buffer of incoming packets and threaded in a non-blocking manner to dissociate matchmaking from the network management load. I ran experiments in a cluster computer composed of dual Xeon X5960 3.46Ghz processors with 124GB of DDR3 memory. Experiments last five minutes in real time and every measure is an average on 20 runs. I set the distance boundaries of the lists to 300 and 50 respectively for algorithms CL and TCL; Subsection 3.5.4 justifies these values.

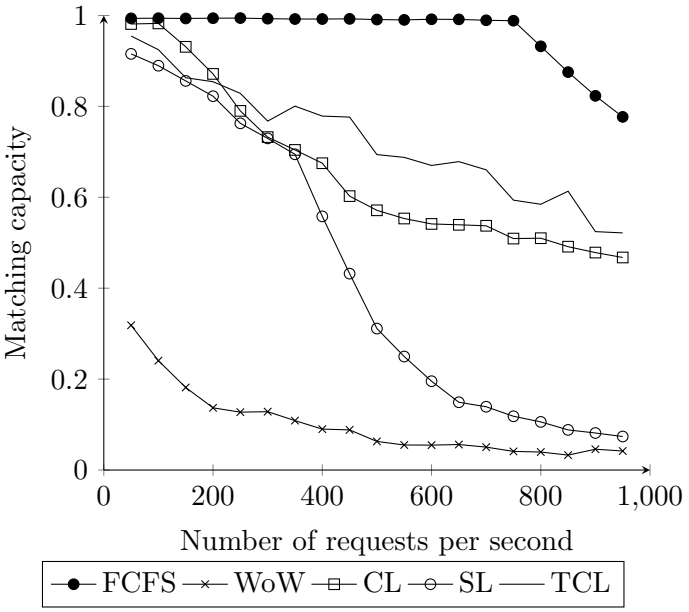


FIGURE 3.14: Comparison of the matching capacities

3.5.1 Matching capacity

In order to analyze how the different matchmaking algorithms hold up to an increasing rate of matching requests, I defined a metric called the *matching capacity*. The matching capacity of an algorithm represents the number of matching requests it successfully processes over a period of 5 minutes. A matching capacity of 1 means that the algorithm successfully matched all client requests received during the 5 minute period, while a value of 0 means that none of the requests got matched.

According to my dataset League of Legends handles a requests insertion rate of 250 requests per second during peak hours, and exhibits a matching capacity of 1 at these times.

Figure 3.14 represents the matching capacity of each algorithm as the requests insertion rate increases up to $1,000$ requests per second. $FCFS$ is my baseline algorithm in these experiments since it is guaranteed to exhibit the best matching capacity (at the cost of the matching distance). WoW and SL don't perform well at all. SL incurs fast degradation when the insertion rate exceeds 400 requests per second, and the matching capacity of WoW is exceedingly low (under 0.36) from the start. Both algorithms suffer from having to manage a single list of requests that grows ever faster with the insertion rate. The particularly poor performance of WoW results from its constant breaking of groups to satisfy others: some requests get pre-matched several times before WoW acheives a final match. Conversely, algorithms that manage multiple shorter lists perform rather well in terms of matching capacity.

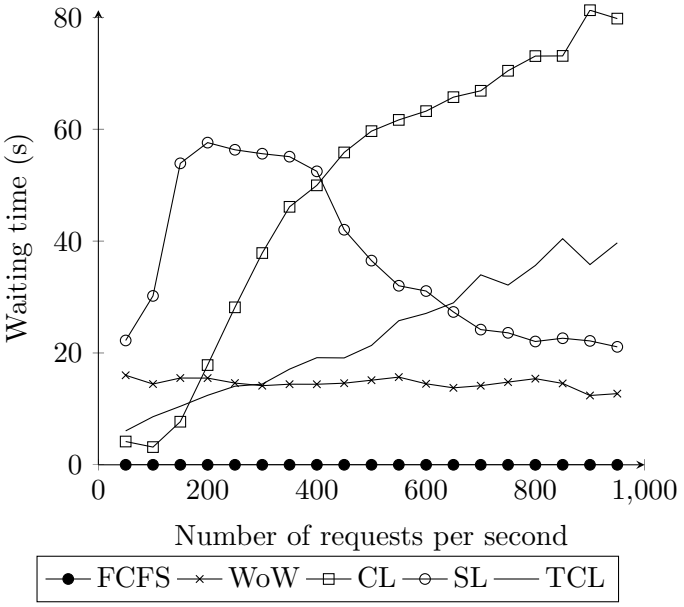


FIGURE 3.15: Comparison of the waiting times

These results indicate that the data structure for indexing matching requests has a strong impact on the matching capacity of a matchmaking approach.

3.5.2 Average waiting time

Figure 3.15 illustrates the impact of increasing insertion rates on the average waiting time. Please note that the plots discard waiting times for requests that never get matched: they are thus strongly associated with the matching capacity plots. For instance the low waiting times for WoW are not very significant given its poor matching capacity. This also explains why waiting times for CL decrease very fast after the initial increase: requests that do get served are fewer and are therefore matched quicker. Another result that is consistent with those of Figure 3.14 is that $FCFS$ keeps outshining all other algorithms at the cost of the matching distance. The most interesting result stems from the comparison between CL and TCL: the matching capacities of both algorithms are very similar, but the introduction of multi-threading significantly reduces waiting times by enhancing the beneficial impact of an indexation in multiple shorter lists.

3.5.3 Matching precision

I finally compare the precision of the matching algorithms by computing the average matching distance they produce. With respect to LoL traces, the matching distance

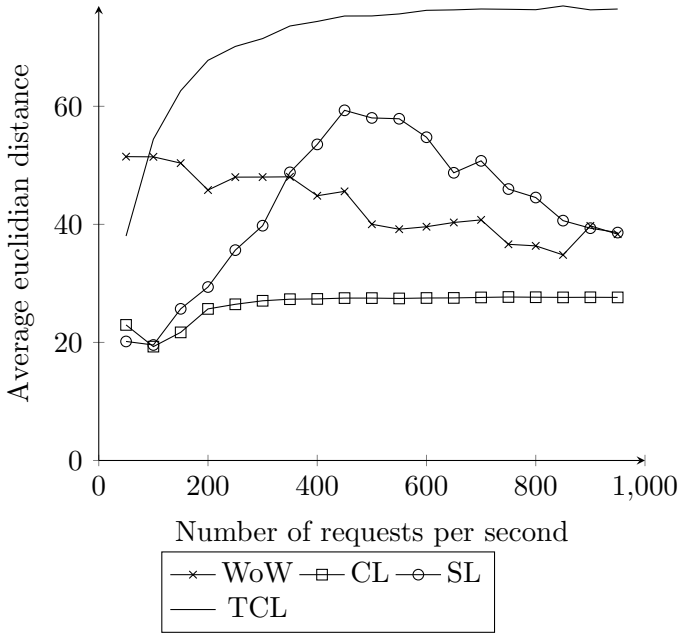


FIGURE 3.16: Comparison of the matching distances

between two players is their bi-dimensional euclidean distance in terms of latency and ranking (see Subsection 3.3.1).

Figure 3.16 shows the impact of increasing insertion rates on the average matching distance. These results only represent requests that got matched eventually, which explains once more why the plots for SL and WoW decrease as the request rate increases. The results of FCFS do not appear in the Figure because they are very high and would hinder the readability of the other plots. Overall, CL and TCL achieve distances well within acceptable ranges, that is they are likely to result in well balanced game sessions.

3.5.4 Adjusting the size of the cutlists

Cutlists algorithms require a phase of testing to find the best suitable size of slice. I ran experiments for both the single-threaded and the multi-threaded version of the cutlists.

Figures 3.17 and 3.18 show that, in terms of matching capacity, the cutlist size has inverse effects on CL and TCL. CL performs better with shorter cutlists, while TCL holds up better to increasing request rates with larger cutlists. This difference in optimal size comes from lock costs when TCL accesses many lists concurrently. Conversely, shorter lists produce faster matches for CL: a result that is consistent with those of SL.

As explained in Sections 3.5.2 and 3.5.3, the matching capacity bears strong impact on the plots associated with waiting times matching precision. Nonetheless, these plots

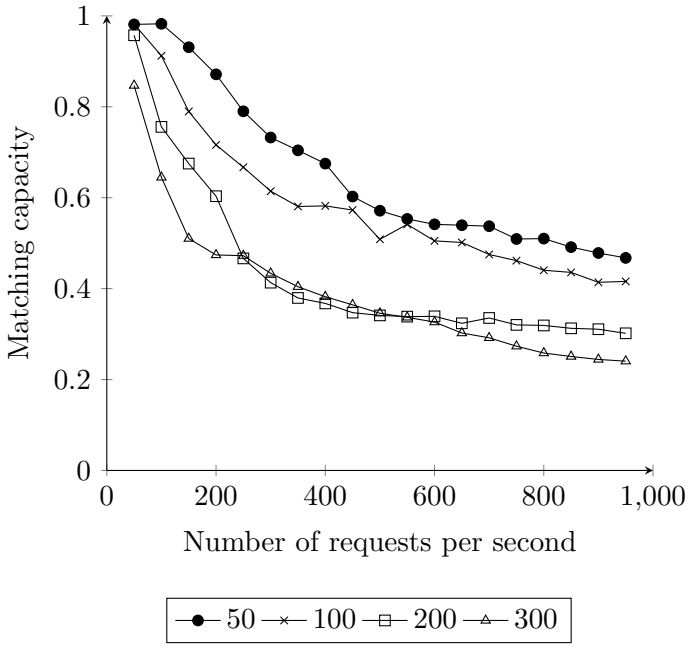


FIGURE 3.17: Impact of the cutlist size on the matching capacity for the single-threaded approach

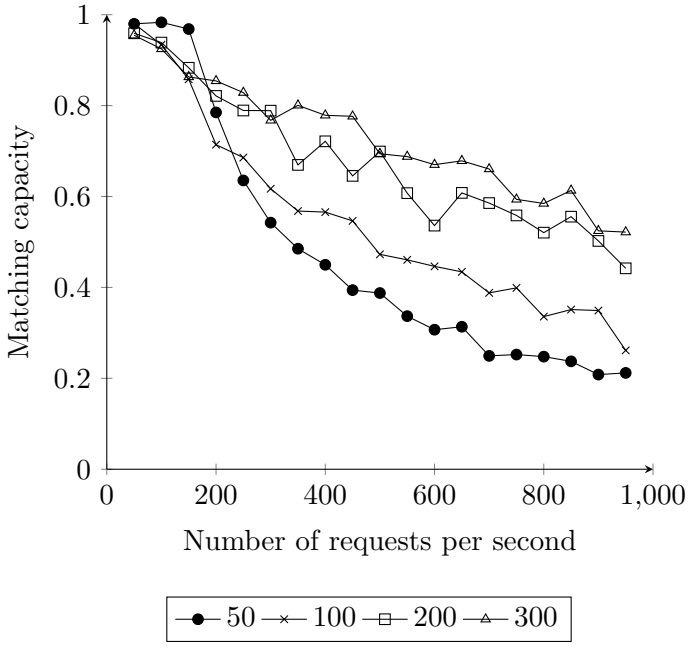


FIGURE 3.18: Impact of the cutlist size on the matching capacity for the multi-threaded approach

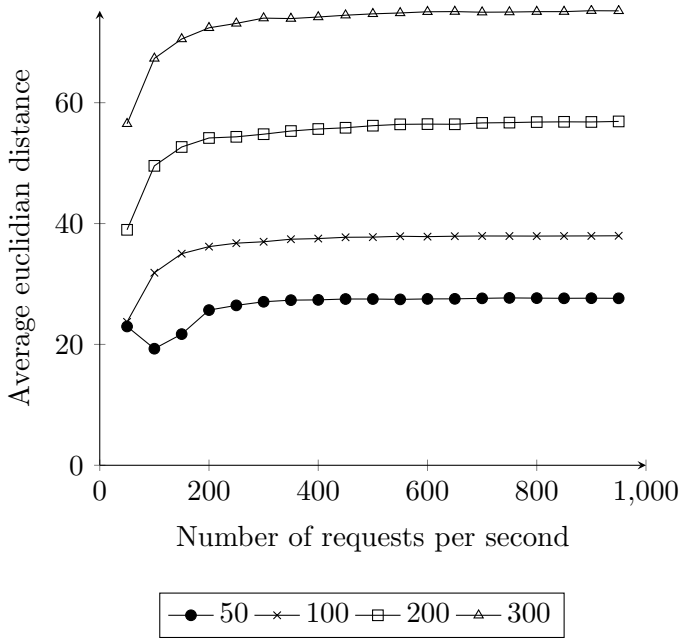


FIGURE 3.19: Impact of the cutlist size on the matching distance for the single-threaded approach

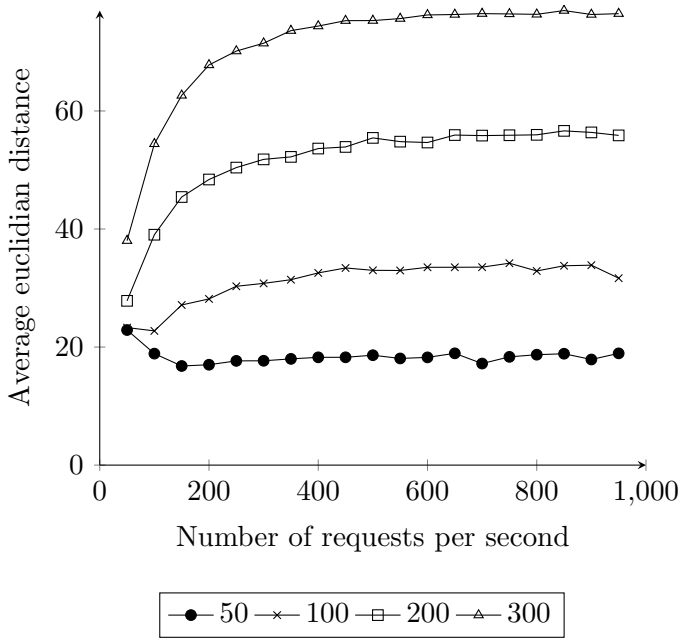


FIGURE 3.20: Impact of the cutlist size on the matching distance for the multi-threaded approach

show that, regardless of the cutlist size, both CL and TCL produce well balanced groups within a decent timeframe.

Figures 3.19 and 3.20 display the respective matching distances acheived by CL and TCL respectively. Despite the increasing request rate and regardless of the cutlist size, neither CL nor TCL ever produces a distance greater than 70. Put in perspective with the optimal distance computed in Section 3.4, this value appears acceptable.

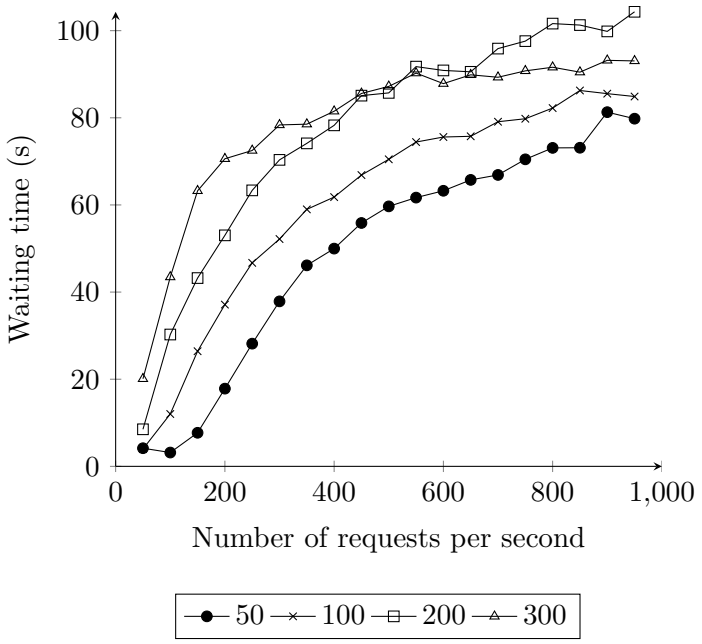


FIGURE 3.21: Impact of the cutlist size on the waiting time for the single-threaded approach

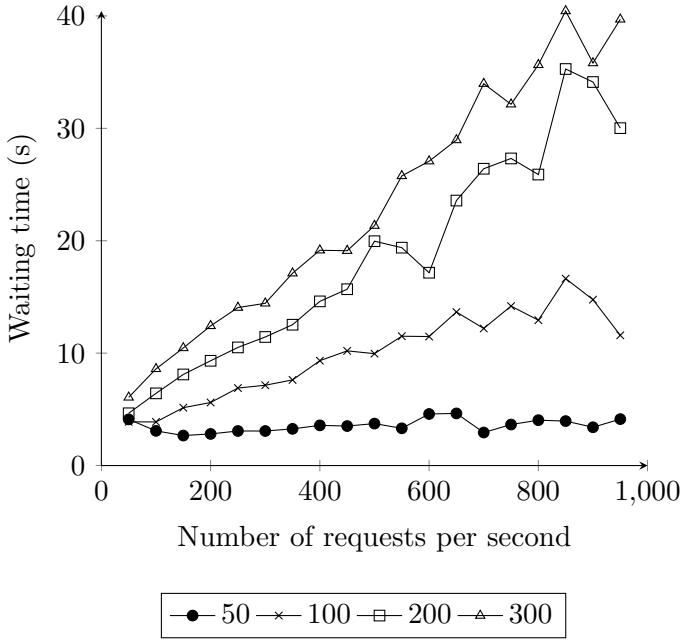


FIGURE 3.22: Impact of the cutlist size on the waiting time for the multi-threaded approach

With respect to waiting times, the results of Figures 3.21 and 3.22 both plead in favor of shorter cutlists.

In the case of CL, this outcome tips the balance towards size 50. Matters are more complex for TCL: it is more important to satisfy a larger proportion of requests if their waiting time remains acceptable. With a waiting time that never exceeds 40 seconds, a cutlist size of 300 appears to be the best for TCL.

3.5.5 P2P scalability and performance

This section describes the performance evaluation of the P2P algorithm separately from the others because I conducted the experiments on a different testbed. All experiments are run on top of PeerSim [80] and involve an increasing number of nodes : 10K, 60K, 100K, 200K, 300K, and 500K. Every node possesses asymmetric connections typical of those obtained through current broadband Internet providers: 800KB/s download and 100KB/s upload. The computing power required by the algorithm on each node is low, so I did not set any CPU limitations.

Every node sends a matching request at the start of the experiment. The work of a node is then circumscribed to routing and comparing messages.

The following experiments show that a P2P matchmaking scales much better than a centralized approach: up to thousands of requests per second. Please note that the request rates are 500 orders of magnitude higher than all previous experiments.

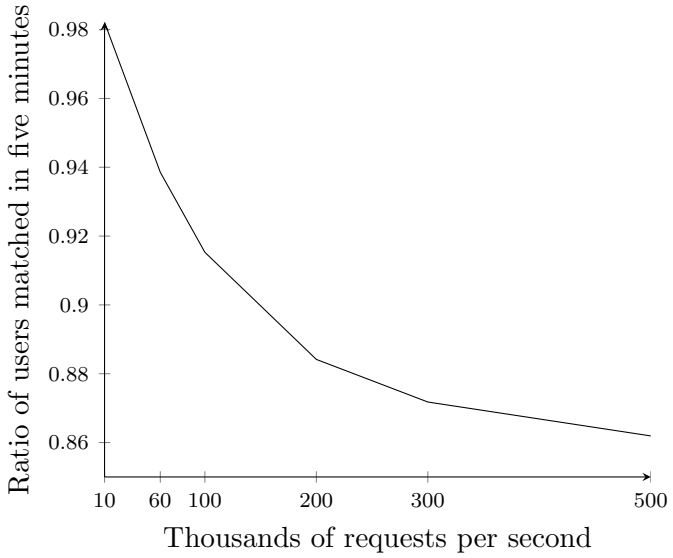


FIGURE 3.23: Matching capacity of the P2P approach

In terms of matching capacity, Figure 3.23 plots the ratio of requests that were matched successfully within 5 minutes as the request rate increases. This ratio starts very high

up and degrades gracefully, remaining above 86% at 500,000 matching requests per second. The degradation is a consequence of the dissemination process: huge burst phases saturate the bandwidth of the nodes.

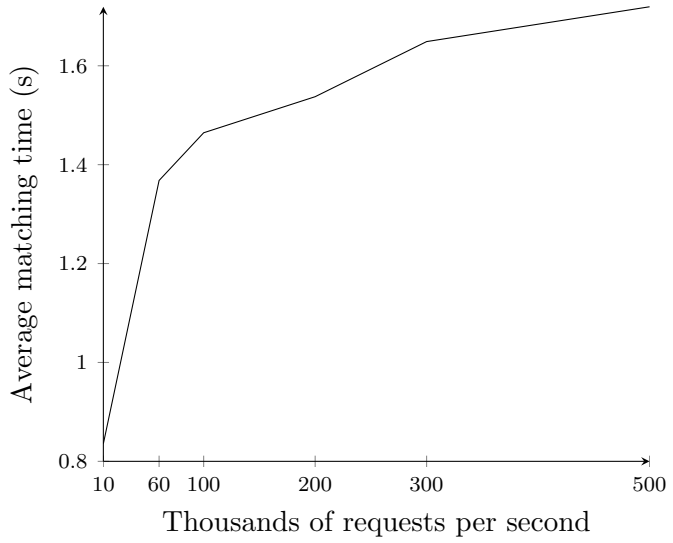


FIGURE 3.24: Matching speed of the P2P approach

P2P also exhibits very high matching speeds (Figure 3.24), and the average waiting time increases logarithmically. Even at 500,000 matching requests per second, it is well under 2 seconds.

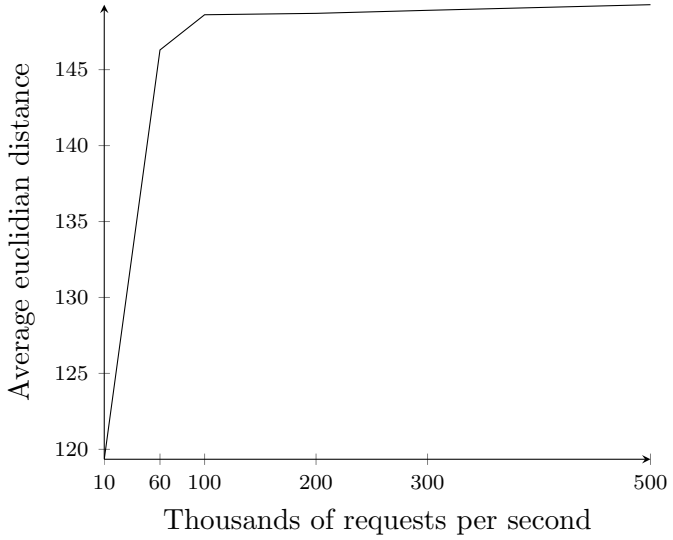


FIGURE 3.25: Matching precision of the P2P approach

In terms of matching distance, Figure 3.25 shows that P2P is not nearly as good as its centralized counterparts. It does achieve a distance that is overall 40% better than that of FCFS, however. Bear in mind that these distances must be mitigated with the high matching capacity: matching a larger proportion of incoming requests makes it harder to approach an optimal match.

Service type	FCFS	WoW	SL	CL	TCL	P2P
Small playerbase	++	+	++	++	++	−
Medium playerbase	++	−−	−	+	+	+
Huge playerbase	++	−−	−−	+	+	++
Fast reply	++	+	−	−	+	++
Low cost	++	+	+	+	−	*N/A*
Accurate	−−	+	+	+	++	+

TABLE 3.1: A comparison of matchmaking algorithms

3.6 Conclusion

User traces enable to improve game software through careful study and analysis. However, game companies seldom make such data available. In this work, I show how to gather a huge quantity of publicly available information about players of a very popular online game. I also analyze this information to draw conclusions about how to improve a critical service for multiplayer online games, namely matchmaking.

I push this analysis further by conceiving several matching algorithms and by comparing their performance in terms of speed, precision, and capacity.

Table 3.1 draws a summarized comparison of the benefits and drawbacks associated with every matchmaking algorithm presented in this Chapter.

FCFS can only be recommended for applications where there is a single criterion: matching speed.

Algorithms which maintain a single list of matching requests can perfectly handle small playerbases, but are not suitable for games that scale. World of Warcraft's scheme of re-breaking formed groups produces finely balanced teams, but drastically impedes throughput.

Approaches that partition matching requests from the start handle throughput very well, and multi-threading speeds up the matching.

Finally a matchmaking service built on top of a P2P overlay ought to be considered. Its matching capacity and matching speed are comparable to those of FCFS, and although there is room for improvement it exhibits a fair matching precision.

I draw two main conclusions. Firstly, matching precision involves many factors beyond waiting time. Secondly, the choice of the matching algorithm has significant impact on the quality of the service and therefore depends heavily on the type of game it supports.

Chapter 4

Scalable cheat prevention for MMOGs

Contents

Massively Multi-player Online Games (MMOGs) aim at gathering an infinite number of players within the same virtual universe. Yet among all of the existing MMOGs, none scales well. By tradition, they rely on centralized client/server (C/S) architectures which impose a limit on the maximum number of players (avatars) and resources that can coexist in any given virtual world [1]. One of the main reasons for such a limitation is the common belief that full decentralization inhibits the prevention of cheating.

Cheat prevention is a key element for the success of a game. A game where cheaters can systematically outplay opponents who follow the rules will quickly become unpopular among the community of players. For this reason, it is important for online game providers to integrate protection against cheaters into their software.

C/S architectures provide good control over the computation on the server side, but in practice they are far from cheat-proof. A recent version of a popular MMOG, namely *Diablo 3*, fell victim to an in-game hack that caused the game's shutdown for an entire day [81]. Not so long ago, someone found a security breach in *World of Warcraft* [82] and proceeded to disrupt the game by executing admin commands. Given that centralized approaches are neither flawless security-wise nor really scalable, there is room for improvement.

This chapter presents a scalable game refereeing architecture. My main contribution is a fairly trustworthy peer to peer (P2P) overlay that delegates game refereeing to the player nodes. It relies on a reputation system to assess node honesty and then discards corrupt referees and malicious players. My secondary contribution is a speedup mechanism for reputation systems which accelerates the reputation-assessment by challenging nodes with fake game requests. It has been published in the Multimedia Systems journal [83].

This chapter is organized as follows. Section 4.1 makes a case for decentralized architectures as building blocks for game softwares. Section 4.2 depicts my main contribution: a decentralized approach for game refereeing that scales easily above 30,000 nodes and allows detecting more than 99.9% of all cheating attempts, even in extremely adverse situations. Section 4.4 outlines the characteristics of the reputation system my solution requires in order to achieve such performance, and details the simple reputation system I designed following this outline. Section 4.5 gives a preliminary performance evaluation obtained by simulating my solution. Finally section 4.6 describe the extended performance evaluation of our platform.

4.1 Scalability issues for cheat prevention

Scalability is a crucial issue for online games; the current generation of MMOGs suffers from a limit on the size of the virtual universe. In the case of online role-playing games, the tendency is to team up in *parties* to improve the odds against other players and the game itself. Being unable to gather a party of avatars in the same virtual world because there are not enough slots left on the server is a fairly frequent cause for frustration among players. The limitation on the scale also has consequences on the size and complexity of a virtual world. Players usually favor games that offer the largest scope of items/characters they can interact with.

Decentralized architectures usually scale far better than C/S architectures. They remove the limitations on the number of concurrent players and on the complexity of the virtual world, or at least relax these limitations significantly.

One solid argument against full decentralization is that cheat detection is very hard to enforce in a distributed context, and it may thus be less effective.

In a C/S architecture, the server side acts as a trustworthy referee as long as the game provider operates it. Let us take an example of a player who tampers with his client to introduce illegal movement commands. This provides an unfair advantage to the cheater as his avatar then acquires the ability to instantly reach places where it cannot be attacked. All it takes for a centralized server to set things right is to check every movement command it receives from player nodes. Such a solution will have a strong impact on the performance of the server, and especially on its ability to scale with respect to the number of concurrent avatars. It is possible to reduce the computational cost of this solution by skipping checks of the received commands. However, it entails that some cheating attempts will succeed eventually.

There are also cheating attempts that a C/S architecture is ill-fitted to address. A player about to lose can cancel a battle by launching a DDOS attack (distributed denial-of-service [84]) on the server and causing its premature shutdown. A decentralized architecture would be far more resilient against such an attack.

In a decentralized architecture, it is not easy to select which node or set of nodes can be trusted enough to handle the refereeing. For instance, let us roll back to the example where a player node sends illegal movement commands. A naïve approach could be to delegate the refereeing to the node of the cheater's opponent. Unfortunately this would introduce a breach: any malicious player node could then discard legitimate commands to their own advantage. As a matter of fact, delegating to any third party node is particularly risky: a malicious referee is even more dangerous than a malicious player.

Reaching a referee decision by consensus among several nodes boosts its reliability. Since it is both hard and costly for any player to control more than one node, the trustworthiness of a decision grows with the number of nodes involved. On the other hand, involving too many nodes in every single decision impacts heavily on performance. Even in a P2P context with no limit on the total number of nodes, waiting for several nodes to reach a decision introduces latency.

4.2 Design of a decentralized refereeing system

The main goal of my approach is to provide efficient means for cheat detection in distributed gaming systems. In this work, I focus on player versus player interactions through the means of fights between them.

The cheat detection mechanism relies on the integration of a large scale monitoring scheme. Every player node participates to the monitoring of game interactions among his neighbor nodes. As such, every node takes on the role of game *referee* and possess an additional entity executing the game engine in order to assess events. To limit collusions among players/referees in order to gain an unfair advantage, no player should have the ability to select a referee. For this purpose, my architecture relies on a reputation system to discriminate honest nodes from dishonest ones. This allows picking referees among the more trustworthy nodes, as detailed in subsection 4.3.2.

4.2.1 System model

Every player possesses a unique game identifier and associates it with a network node, typically the computer on which the player is running the game software. Every node runs the same game engine: the code that defines the world, rules, and protocols of the game.

An avatar represents a player within the game world. Data describing the dynamic status of the avatar is called the player state; it is stored locally on the player's node and replicated on other peers to prevent its corruption.

Players will interact with each other by the mean of battles. Battles are a succession of events that lead to a battle conclusion with one winner. Players will emit alternatively a state event when they want to change their own state and an action event when they want to impact the other player. In order to check for cheating attempts, players event will pass through referees. Figure 4.1 depict this particular process between players and referees that aims at reproducing in each referee the behavior of an MMOG server.

The player "Node 1" here is the initiating player of the battle. He will request first an opponent from his player list to fight with him. If the player is agreeing, the initiator node then ask a trusted referee from his list of referees to assess the fight. Once the referee accepts, the initiator node propose the referee to the other player. If both players agree on the same trusted referee, the battle can start. Events are sent until one of the two player life points reach 0 and can be declared loser of the battle by the referee. Each event sent is checked by the referee and the referee is the only authority allowed to make communications with players during the battle. If a player attempt to bypass this rule, he will be seen as incorrect by the other player client.

There is no limit on the total number of player nodes in the system. Player states are stored/replicated on network nodes and every player node maintains a list of geographical neighbors. The game downloading phase is considered complete: all static game content is installed on every node. All data exchanges are asynchronous. Nodes communicate by messages only; they do not share any memory.

I assume that all shared data to the game and the virtual universe can be accessed in an RTT between the data location and the node. Systems such as [22, 23, 85] already address this issue.

Both the player and the referee codes and protocols can be edited by third parties: my approach aims at detecting and addressing such tampering. The matchmaking system, the part of the game software that selects opponents for a battle, falls out of the scope of this work. Leaving it on a centralized server would have little or no effect on the overall performance, and I assume it to be trustworthy.

4.2.2 Failure model

My distributed approach aims at offering a secure gaming protocol, so as to detect every corrupted game content that passes through the network and impacts other players. My goal is to ensure the lowest cheat rate even when possible failures occur.

Section 2.3 describes different type of cheating attempts that can be found in MMOGs. I set the system to rely on software based cheat detection and will use the computing power residing in nodes in order to assess players transactions.

I want my distributed approach to at least match the degree of protection guaranteed by traditional C/S architectures. For this purpose my solution addresses all of the following failures :

- delays or modifications of the communication protocol,

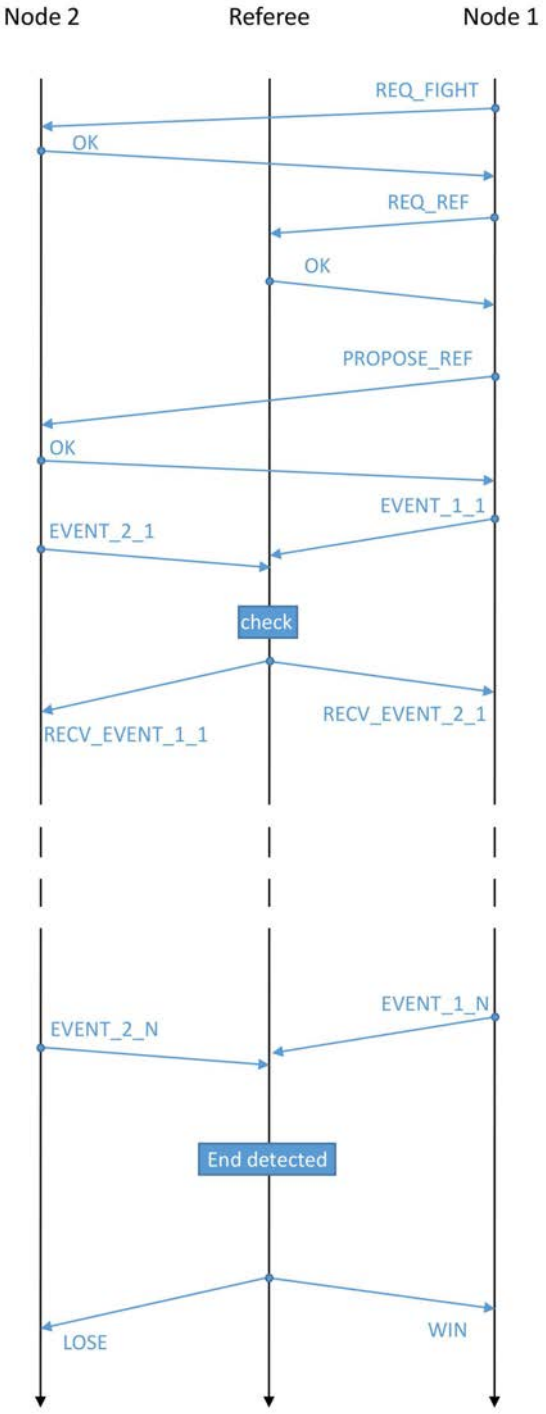


FIGURE 4.1: Player/referee interactions during battles

- modifications of stored data,
- denial of service attacks,
- collusions between nodes as long as there is at least a majority of correct referees per battle.

In order to scale, my decentralized architecture requires trustworthy nodes to act as referees. It is risky to confer referee statuses indiscriminately, as it provides an easy way for dishonest nodes to turn the game to their advantage. Hence my solution relies on the availability of a distributed reputation system [59] in order to identify the most trustworthy nodes. Any such system [58, 60] would work, as long as it collects feedback about node behaviors and computes values that describe these behaviors.

4.2.3 Architecture model

Every node is part of two overlays, and thus maintains two neighbor lists: a list of the player's immediate neighbors within the virtual world of the game, and a list of neighbor nodes within the reputation P2P overlay.

The first overlay is loaded at startup and link players together with a number of neighbors following a powerlaw distribution. The second is built over time with player interactions: by gathering reputation values from the referees transactions, players fill their list of available referees. In the rest of this work, I use the term *neighborhood* to refer to both lists.

4.3 Distributed refereeing protocol

My decentralized refereeing approach starts with an initialization phase that allows the reputation system to complete a first estimation of node behaviors. Afterwards, any node can decide to initiate a battle with any other node, and asks one or more referees to help arbitrate the outcome. A battle between two opponents contain a series of event exchange until a victory occurs. During a battle, every referee monitors the game commands sent by both opponents, and then decides the outcome of the battle based on the game data and on the correctness of the commands.

4.3.1 Node supervision

When the reputation system is ready, players can initiate "real" battles under the supervision of one or more referees. Section 4.4.3 details the reputation initialization issues.

Once a battle has been initiated, the opponent nodes can create events they send to referees. Those events are based on the current player states. There are two types of events : *actions* describing inputs and *states* containing player states.

In my model, a cheater is a player node which either (a) makes up an event which does not match its state according to the game engine, or (b) delays the emission of an event.

Upon receiving an event from a player, a referee checks if the event is valid before transferring it to the player's opponent. Events are thus exchanged between two opponents under the supervision of one or more referees,until the battle ends. Three possible conditions lead to the end of a battle : (i) one of the player health drops to 0, there is a winner; (ii) one or both players are cheating, the battle is canceled; or (iii) both players agrees to call it a draw. If a cheater attempts to declare itself a winner illegally, both the referees and the opponent will detect his attempt.

4.3.2 Referee selection

In order to optimize cheat detection, the system picks referees among the trustworthy nodes in the list of available referees. As described in Section 2.5, trustworthy nodes are those whose reputation is above a fixed threshold T. A node A will only consider another node B as a potential referee if the reputation value associated with B as a referee increases above T. Self-refereeing is prohibited for obvious reasons: therefore player nodes are excluded from the referee selection for battles they are involved in.

I designed my referee selection mechanism to reduce the control any single node can have over battles. A single player node will have a low probability of managing to impose referee nodes. To achieve this goal, two player nodes involved in a battle must agree on the selection of referees. The player that initiates the battle first searches for available referees and books them for a battle. The player will then suggest this list of referees to its opponent. The opponent then double checks that those referees are trustworthy. A node that makes a lot of incorrect selections will quickly go down in the node reputation values, thus leading to its detection and possible exclusion from the system.

Since reputation values are dynamic, it is possible for a referee to lose its trustworthiness in the middle of a battle. In this case, the battle is canceled: all refereeing requests associated with the corrupt referee are tagged as fakes and will only be used as information for the reputation system.

4.3.3 Cheat detection

Every battle is composed of an event stream which one of the opponents may try to corrupt. The refereeing system verifies events as follows.

Any player node creates events based on the game engine. Event can be of three types: state event, action event and victory event. In order to change the state of his avatar, a player node must issue a refereeing request containing an event description called action. In reply to every action emitted, a state event will be required to verify that the player applied the action fairly. Finally, when the battle ends, a victory event is sent to the referee for verification.

Upon receiving the two events –one for each opponent–, a referee will use the game engine to verify:

- the initial state sent by the players and their consistency with the replicas stored in the P2P overlay;
- every action, to assess whether it is coherent with the player state by replaying it on the referee;
- the new state of every player, to ensure that the actions got applied by comparing with the result of the replay done on the referee;
- victory announcements if any, to prevent the most obvious cheating attempts.

Overall, the protocol consist on a redirection of the messages that usually arrive on a centralized server towards nodes of the network. This set of verifications are not as intrusive for a game as they might seem.

A battle triggers a loop of verifications on the referee (Figure 4.2); one verification per skirmish, until a victory occurs or until the battle gets canceled.

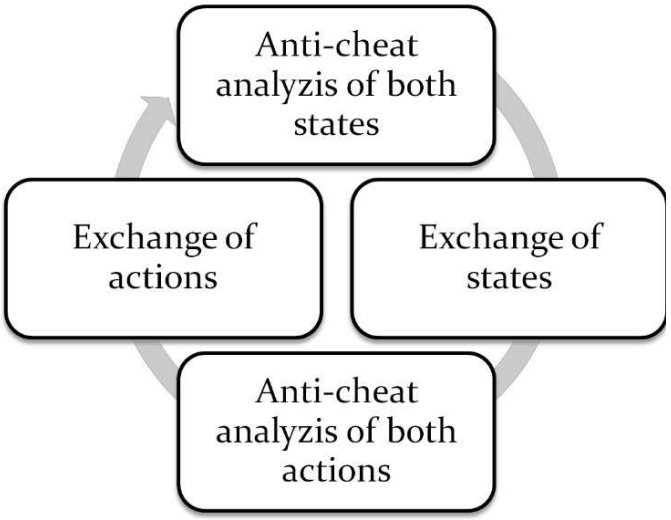


FIGURE 4.2: Referee automaton used to analyze a full battle.

Referees for a same battle send their decisions back to the player nodes directly; they don't communicate to reach a consensus. This does not introduce a breach in my security, as I can detect incorrect nodes while they try to cheat on decisions.

My architecture systematically detects cheating attempts, whether they come from a malicious player or from a malicious referee:

If one of several referees sends a wrong decision to the players, the players will detect the inconsistency by observing which referee is a minority. This protect from incorrect referees emitting influenced decisions as long as we have a majority of correct referees per fight.

If an incorrect player wants to cheat and decides to take into account a wrong decision, correct referees will detect an incorrect player state at the next iteration. This protects from arbitrary player choices.

Finally, if an incorrect node turn an event into a victory event incorrectly, both its opponent and the referees will consider it as malicious.

4.3.4 Multiplying referees to improve cheat detection

One referee isn't enough to ensure that my approach is sufficiently cheat-proof. A malicious node can temporarily send correct responses to gain a good reputation, and then issue corrupt decisions if it manages to acquire a legitimate referee status. Associating more than one referee with the same battle counters such behaviors.

My solution enables players to select N referees for the same battle, with N an odd number. This is done to obtain a majority over the referees replies. Once they have agreed on the referee selection, the players submit their requests concurrently to every referee. It is considered that using our reputation system, $\frac{N}{2}+1$ referees will be correct in most of the battles (see Section 4.5.5 for cheat detection ratio details). A player that receives $\frac{N}{2}+1$ identical replies may then consider the result as trustworthy. Figure 4.3 represents the overall communication links the two players will have with their N shared referees.

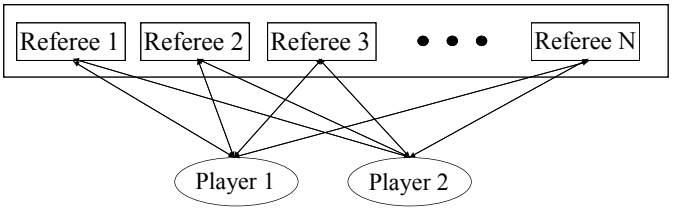


FIGURE 4.3: N-referee configuration

This approach strengthens the trustworthiness of the arbitration, since the probability of picking $\frac{N}{2}$ malicious referees at the same time is considerably lower than that of selecting a single malicious referee. At the same time, it improves the detection of both malicious players and malicious referees and helps preventing collusions between a player and a referee. As referees assess players behaviors adding more of them creates more interactions and therefor more reputation values which speeds up detection. Collusion is a costly strategy, and the cost grows with the number of nodes involved. This is enforced my approach since:

- a player alone cannot influence the selection of the referees,
- colluders must first work to obtain good reputations before starting to cheat,
- a node can never know whether it is handling a legitimate or a fake refereeing request with the testing system (see Section 4.4.3).

I analyzed the impact of the number of referees on the efficiency of the cheat detection and on the overhead. The results of this analysis, along with other results, are presented in section 5.3.

Avoiding collusions for a given battle requires an assessment by a majority of correct referees (correct referees do not collude). Collusions among referees, or among players, or even between referees and players will result in a detection: (i) players will always receive a majority of correct referee decisions as there is a majority of correct referees by assumption, different decisions come from faulty referees; (ii) correct referees detect players that do not respect their decisions when they receive further messages from both players; (iii) the majority of correct referees will detect collusions between players and referees and updates of the virtual world data remain reliable anyway as they require a majority of referees. Thus, no possible configuration provides the ability to undergo an illegal action as long as, for each battle, a majority of referees are correct. Notice that if there is a majority of malicious referees for a given battle, they might behave badly, however this can be detected by correct players, leading to a reputation decrease for the referees.

4.4 Reputation management

In order to scale, the decentralized architecture requires nodes to act as referees. It is risky to confer the referee status to every node indiscriminately, as it provides an easy

way for dishonest nodes to turn the game to their advantage. Therefore my approach requires a way to pick out nodes that are reliable enough to act as referees.

My solution integrates a distributed reputation system to identify the most trustworthy nodes. A reputation system [59] aims to collect and compute feedback about node behaviors. Feedback is subjective and obtained from past interactions between nodes, yet gathering feedback about all the interactions associated with one node produces a relatively precise opinion about its behavior in the network. This allows to detect players that cheat and to avoid potentially malicious referees.

There are two levels of trustworthiness for every node: as a player within the game (*player reputation*), and as a referee for the game (*referee reputation*). I dissociate both levels entirely as a node play two roles in our system : the player, and the referee. A node can be correct as a player but have his machine infected therefor issuing incorrect referee requests. This would penalize correct players by removing them the capability to play. With a reputation value for players and referees the system allows a finer assessment as to which entity of a node is faulty.

4.4.1 Assessment of the reputation

Every node stores a local estimation of the *player reputation* and of the *referee reputation* associated with every node in its neighborhood. Given the failure model, it makes no sense to fully trust a remote node, and therefore a reputation value can never equal the maximum value. A reputation value belongs to $[0, 1000[$. Value 0 represents a node which cannot be trusted, whereas the reputation value of a very trustworthy node approaches the unreachable 1000. Initially, when assessing a node that has no known history as a player (respectively as a referee), its *player reputation* (resp. *referee reputation*) value is set to 0. They will be chosen as the last available nodes.

There are two types of direct interaction between nodes that lead to a reputation assessment. A refereeing request causes the referee node to assess the *player reputation* of the requester, and the player node to assess the *referee reputation* of the requested. The referee selection process triggers a mutual *player reputation* assessment between nodes.

Every time a node interacts with another node, both nodes assess the outcome of the interaction and update their local value for the reputation of their counterpart. A valid outcome increases this value (*reward*), while an incorrect outcome will decrease it (*punishment*). Punishments must always have a greater impact on the reputation value than rewards. This prevents occasional cheaters from working their way to a good

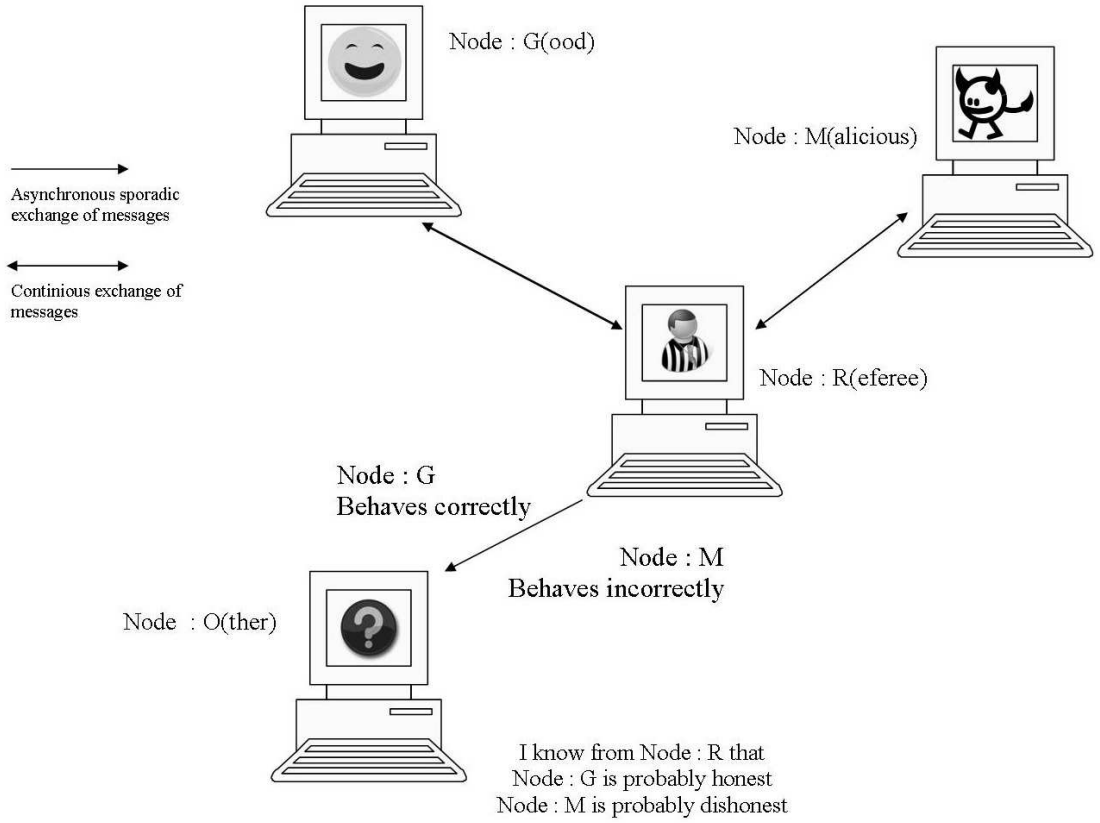


FIGURE 4.4: Global reputation assessment in the network

reputation value, which in turn confers an advantageous position for avoiding cheat detection or for acquiring referee status.

Evaluating a reputation through direct interactions only is a bad idea. Firstly it means that, in order to consolidate its reputation assessment, every single node must carry out multiple transactions with each other node it wants an opinion about: it is costly both in terms of time and resources. Secondly a malicious node may act honestly with a restricted set of nodes in order to escape banishment from the game if it gets detected by other nodes.

Nodes exchange their local reputation assessments for both players and referees sporadically to their neighborhood to help build a common view of their neighborhood in terms of player/referee trustworthiness. Reputation values are then carried to the neighbors, which will apply them locally, and spread them in the next iteration. Reputation updates resulting from an incorrect interaction are sent immediately while those that sanction honest behaviors are delayed until the end of the battle. This saves bandwidth and prevents reputation increases for nodes that exhibit an erratic behavior with respect to trustworthiness.

Figure 4.4 gives an example of the way reputation assessments get propagated among nodes. Node $O(ther)$ starts interacting with node $R(eferee)$ to get information about a node it intends to fight. It so happens that node R is currently refereeing a battle between nodes $M(alicious)$ and $G(ood)$, and has therefore assessed their reputations. Node R piggybacks its assessments on its messages to node O.

The indirect reputation assessment of a node A by another node B generally relies on three types of information:

1. the current reputation value that B associates with A,
2. the evolution of the behavior of A as perceived by B,
3. and the recent opinions that B or other nodes may express about A.

Upon receiving fresh data about the behavior of a node, reputation systems such as TrustGuard [86] use a *PID (proportional-integral-derivative)* formula on these three pieces of information to compute a new reputation value. The principle of a PID formula is to carry out a weighted sum of the local reputation value at $t-1$, of the integral of the local reputation values since the system startup, and of the differential with newly received reputation values:

$$R(t+1) = \alpha * R(t) + \beta * \frac{1}{t} * \int_0^t R(t)\,dt + \gamma * \frac{d}{dt} * R(t)$$

The values for parameters α, β and γ are totally dependent on the application. A high value for α will confer a greater importance to past reputation values stored locally. This is useful in systems where close neighbors cannot be trusted. Parameter β focuses on the consistency of the behavior of a node. Systems with a high value for β prevent malicious nodes from wiping their slate clean with a few honest transactions. Finally, parameter γ reflects the transitivity of the reputation, in other words the direct impact of a new opinion on the local assessment. A high value for γ implies that the local reputation value of a node will be more sensitive to new values expressed locally or by other nodes.

My reputation system simplifies this computational model to facilitate its implementation. I achieve this simplification by transforming the model into an arithmetic progression. Let $I(t)$ be the value of the integral at time t, $D(t)$ the value of the differential at time t, and $a(t)$ a reputation value received from another node at time t.

My first transformation is to reduce the integral to an iterative average:

$$I(t) = \frac{I(t-1)+a(t)}{2}$$

This transformation adds a fading factor to the past behavior of the evaluated node. Most reputation systems introduce a similar factor to increase the impact of the recent behavior, and therefore to be more responsive to sudden changes in the behavior of the evaluated node.

The second transformation reduces the derivative to a difference between two values:

$$D(t) = a(t) - R(t)$$

The computation for $R(t)$ in my implementation thus breaks down to:

$$R(t+1) = \alpha * R(t) + \beta * I(t) + \gamma * (a(t) - R(t))$$

In terms of reputation assessment, the requirements of my refereeing architecture are very basic. I need every node to store subjective values about the behaviors of the other nodes in their neighborhood. With this in mind, I designed my own reputation system because it was simpler to prototype it quickly and integrate it into my simulations. Other reputation system with similar characteristics [58, 60] would be able to get adapted to my architecture.

4.4.2 Parameters associated with my reputation system

Several parameters allow to adapt the reputation system to the requirements of the application. Setting the parameter values is closely related to player/referee behaviors that are specific to every game. Hence the fine tuning of these values requires extensive benchmarking during the test phase of the game software. Nevertheless, there are some pointers for the parameterization of the reputation system in a gaming context.

The first obvious set of parameters α, β, γ characterizes every reputation assessment. As mentioned earlier, in the context of cheat detection The system should focus on its reactivity to incorrect actions. As the main component of the formula for this purpose, γ ought to be set to a high value.

Values υ (up) and δ (down) correspond respectively to rewards and punishments. As mentioned earlier, setting δ significantly superior to υ discourages malicious behaviors and prevents the promotion of corrupt nodes to referee status.

Every local reputation value associated with neighboring nodes is reset after ρ updates; ρ is a parameter of the game code, as it depends on the robustness of the game design. This reset strategy helps prevent dishonest nodes from earning a good reputation, and

at the same time protects honest nodes from bad recommendations spread by dishonest nodes.

I determine whether a node acts honestly by enforcing a global threshold T on reputation values. This threshold has two main uses:

- it serves as the main metric for the referee selection mechanism described in Subsection 4.3.2,
- it reduces the overall CPU load by randomly skipping refereeing requests from reputable nodes, as described in Subsection 4.4.4.

Similarly to ρ, the value of T is highly dependent on the game design; it must be set to the best trade-off between cheat detection and efficiency. If the game software designers expect a proportion of cheaters that is extremely high, then the threshold value must be set very high. Indeed in such cases the reputation of incorrect nodes will be close to that of correct nodes, as few battles will be able to finish and cheaters will be harder to detect. Therefore a high threshold value will ensure that fewer malicious nodes will pass through the cheat detection. Conversely, a system that encounters a moderate proportion of cheaters will witness the emergence of two distinct reputation value averages among nodes : one average for honest nodes and another average for dishonest ones. The threshold becomes easier to set, as its value can be chosen somewhere between both averages.

4.4.3 Jump start using tests

Identifying trustworthy nodes is particularly tricky in two specific situations:

1. during the game startup,
2. and when a new node joins the application.

In both cases, reliable reputation values cannot be assessed for lack of transactions in sufficient number to study node behaviors.

I solve this issue by integrating both *fake testing* and an *initialization phase*.

Fake testing consists in hiding fake requests into the flow of legitimate refereeing requests. Fake requests are identical to legitimate requests, and therefore recipient nodes cannot distinguish between both types. When two player nodes fail to book enough trustworthy referees for a real battle arbitration, they initiate a referee testing phase

and pick a currently untrustworthy node in their neighborhood. The testing phase consists in sending requests associated with a normal battle, according to the following rules.

1. Requests contain both incorrect and correct actions/states.
2. Player nodes verify the referee answers to every action/state request.
3. Player nodes maintain a ratio of correct/incorrect event requests high enough to protect their own player reputation.

Since the tested referee cannot discern test requests, it may suspect the requesting nodes of being malicious and cause their player reputation value to be reset to 0. Rule 3 avoids this situation by balancing incorrect test requests with correct ones.

The testing phase lasts until the referee reputation value allows one of two possible decisions: either the launch of a real battle with the tested node as referee, or a reputation message is issued to neighbors to notify that the tested node is suspected unfit for refereeing. At that point, both player nodes notify the referee to cancel the battle. All games provide a surrender procedure: It is used here as a mean to end the testing phase in such a way that it remains undetectable for the referee.

New nodes start with an *initialization phase* where they only send out fake requests to their neighbor nodes in order to fill its referee list. A node ends its *initialization phase* as soon as it has identified enough potential referees to start a battle – the minimum number of referees required for a battle is discussed in Section 4.5. The *initialization phase* has a very negative impact on the *player reputation* of the node as it sends both correct and incorrect commands to test for potential referees. The system compensates for this by enforcing a full reset of the reputation value periodically, as described in Subsection 4.4.2.

New nodes also incur a high probability of getting tested when entering the system as they arrive in an available state. Malicious new nodes may use this knowledge to their own advantage by acting honestly for a long period of time and then cheating. This strategy will be short-lived since no node is ever fully trusted: a correct node will inflict a heavy punishment as soon as it detects a malicious interaction. Moreover the fake testing accelerates the reputation assessment among nodes, and also discourages cheating attempts: tampering is already risky, why try it for potentially no benefit if my reputation value will be destroyed later? A variant strategy, reentrance to regain a clean slate, is not valid as each player node is associated with a unique game identifier.

Fake testing remains effective for the whole game duration. Besides accelerating the detection of malicious nodes and the integration of honest nodes, it maintains a list of potential referees in the neighborhood of every node. This list is used when the most trustworthy neighbors are busy refereeing other battles. Maintaining this list also offers redemption to correct nodes that got their reputation spoiled by malicious nodes, or to nodes that exhibited an incorrect behavior because of a latency spike during a battle as they are never excluded from the list. With the frequent resets of reputation value, a good behavior will raise their position in the list.

4.4.4 Reducing the overhead induced by the cheat detection

Refereeing is a costly mechanism in terms of CPU and network usage as referees have to frequently replay players events. In order to reduce these costs, C/S architectures introduce heuristics aimed towards skipping some refereeing requests. My solution extends this idea by identifying situations where skipping requests is more logical, that is when the request comes from a node with a good *player reputation*.

I use the following formula to decide how often refereeing requests can be skipped:

$$SkipRatio = \frac{R-T}{V-T+1}$$

With R the reputation value of the requesting node, T the threshold value and V the maximum reputation value in the system.

SkipRatio will increase as the reputation value of the requesting node gets closer to the maximum value. The threshold guarantees that requests from trustworthy players are the only ones that get skipped. I added the plus one to the denominator in order to ensure that even the most trustworthy node requests get tested once in a while. As reputation values grow very slowly, a node can only get to this point if it has acted honestly towards a significant number of other nodes for a long time.

4.5 Performance evaluation

This Section details the results of the performance evaluation I conducted in order to check the scalability and efficiency of my solution. I base my evaluation on simulations in order be able to analyze the behavior of my system when thousands of nodes are involved.

My distributed refereeing system aims at offering an alternative to the C/S architecture in the MMOG context. To prove that my approach is worthwhile, the overhead generated by my solution must be kept low in order to allow scalability and to offer a user experience that is indistinguishable from classic C/S performances. Obviously, I also expect my solution to detect the highest possible number of cheating attempts.

4.5.1 Simulation setup and parameters

I use the discrete event simulation engine of PeerSim [80] to conduct simulations of my system. The present performance evaluation constitutes a first assessment of feasibility. I intend to validate my solution further by plugging it over a P2P gaming overlay with a *world* component later on, but *virtual world* management falls out of the scope of this work.

My simulation aims at reproducing the topology of real game overlays: a node distribution with zones of varying densities following a powerlaw. Player states, accordingly to what I expect to find in most MMORPG games, are all different. Player levels differ, producing both short and long battles. The world is infinite: players probe their neighborhood every five minutes to find opponents, and I set the average battle duration at two and a half minutes. In the extended performance section, this time will be extended to 15 minutes in order to test longer battles. Battle related messages are sent when a player emits an action. Inputs are polled 25 times per second to respect a decent frame rate. Other messages such as testing messages and reputation messages are sporadic. Reputation messages are sent either when a battle ended without incurring any incorrect action, or when a node tried to input too many incorrect actions. To prevent side effects and cyclic behaviors produced by the fixed periodicity of battles, I introduced skewness by ensuring that waiting times between battles, fighting states initial values, inputs, and so on, differ for every node.

Every run simulates game interactions among 30,000 nodes over a 24 hours period, and uses random seeds provided by PeerSim to generate original player states. Every measure presented hereafter is a mean value computed with results from 40 different simulation runs.

I performed my simulations on a Dual-CPU Intel Xeon X5690 running Debian wheezy v3.2.0-4 at 3.47Ghz, with 128GB of available memory. Every run generated an average of 24 CPU threads.

In order to find appropriate values for the parameters of my reputation system, I ran several simulations prior to the ones presented in this Section. I found the optimal

parameter setting to be as follows: threshold $T = 300$, reset frequency $\rho = 50$, $\alpha = 0.2$, $\beta = 0.2$, and $\gamma = 0.8$. γ is much higher than α and β because I want the detection mechanism to focus on quick reactions to the strong punishments that incorrect nodes can receive.

Punishment δ and reward υ produce specific behaviors in my reputation system. δ forces the reputation value to converge quickly towards 50 when small errors are detected. Multiple preliminary simulations yield 50 as the average reputation value an untrustworthy node achieves on the long run in my system. Conversely, υ enforces a slower convergence of the reputation value for honest nodes towards 500, the overall average for trustworthy nodes.

I set the concurrent number of nodes in my system to the largest value my simulator could handle while running on my hardware: 30,000 nodes. It still is twenty times as big as the unofficial client/server concurrent limit I identified through various Internet sources[87, 88]. I also ran some short simulations with up to 60,000 nodes, and observed no difference in the behavior of my system.

The metrics I used to assess the scalability and efficiency of my system are:

- the latency,
- the network bandwidth consumption,
- the CPU overhead introduced by my architecture,
- the percentage of undetected cheating occurrences.

4.5.2 Latency

I introduce a uniformly random latency for every exchange between two nodes and set the lower and upper bounds to 10 and 40 ms respectively. I first estimated these values by monitoring the traffic I generated while playing *Guild Wars 2*, and later backed my estimations by interrogating a database I have put together with data acquired from a *League of Legends* server, and covering over 20 million games.

4.5.3 Bandwidth consumption

Similarly to latency, I evaluated the bandwidth consumption of my solution per message. I kept count of the number and the size of all the messages exchanged during every

simulation run. With these logs, I correlated message contents and real memory usage of data such as integers or floats in classic games to compute realistic message sizes.

There are two cases: state messages and action messages. In my scenario, state messages contain all the game related information a node knows about itself. This amounts to 84 bytes when serialized before being sent on the network, and matches the value found in the literature [89]. Action messages are a bit smaller (16 bytes when serialized), as they only contain incremental information. By correlating these values with the message count of every node, I deduce the average bandwidth used on a single node over time.

As shown in Figure 4.5, each node (player plus referee) consumes a mean of 4KB/s once the system has stabilized. Given that the maximum consumption never exceeds 8KB/s, the bandwidth consumption is very low.

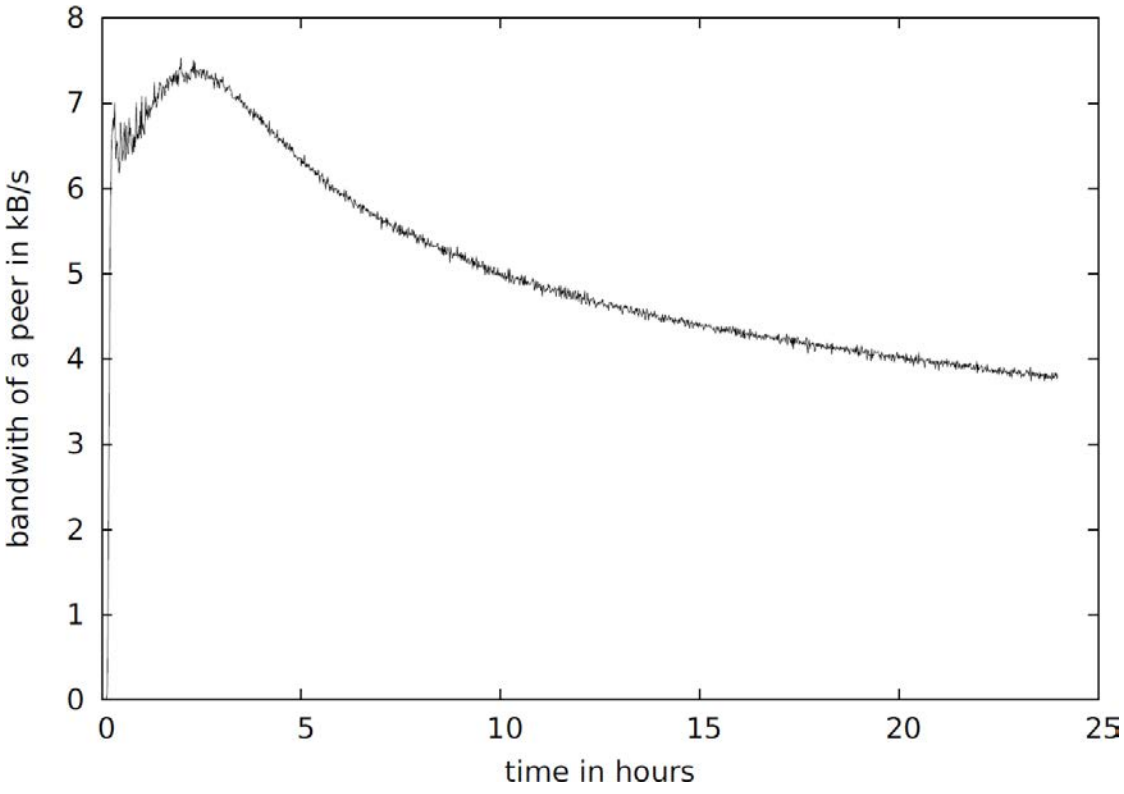


FIGURE 4.5: P2P node bandwidth usage over time

I also performed a theoretical comparison between my approach and the client/server approach. For this purpose, I defined a virtual centralized server as a computer with unlimited resources, able of handling all concurrent requests without crashing. I then summed up the average workloads handled by every peer in my distributed solution during the simulation runs and divide it by the number of referees I run in the decentralized version, with three referees per battle. Figure 4.6 shows that the total bandwidth used peak in the client/server architecture reaches 67MB/s. With this load in a real game

deployment, the server would crash and the players would be disconnected. Such situations occur pretty frequently with existing games nowadays. A cluster of servers might be able to handle this direct overload, but then comes the need to put in place a complex coordination between the servers.

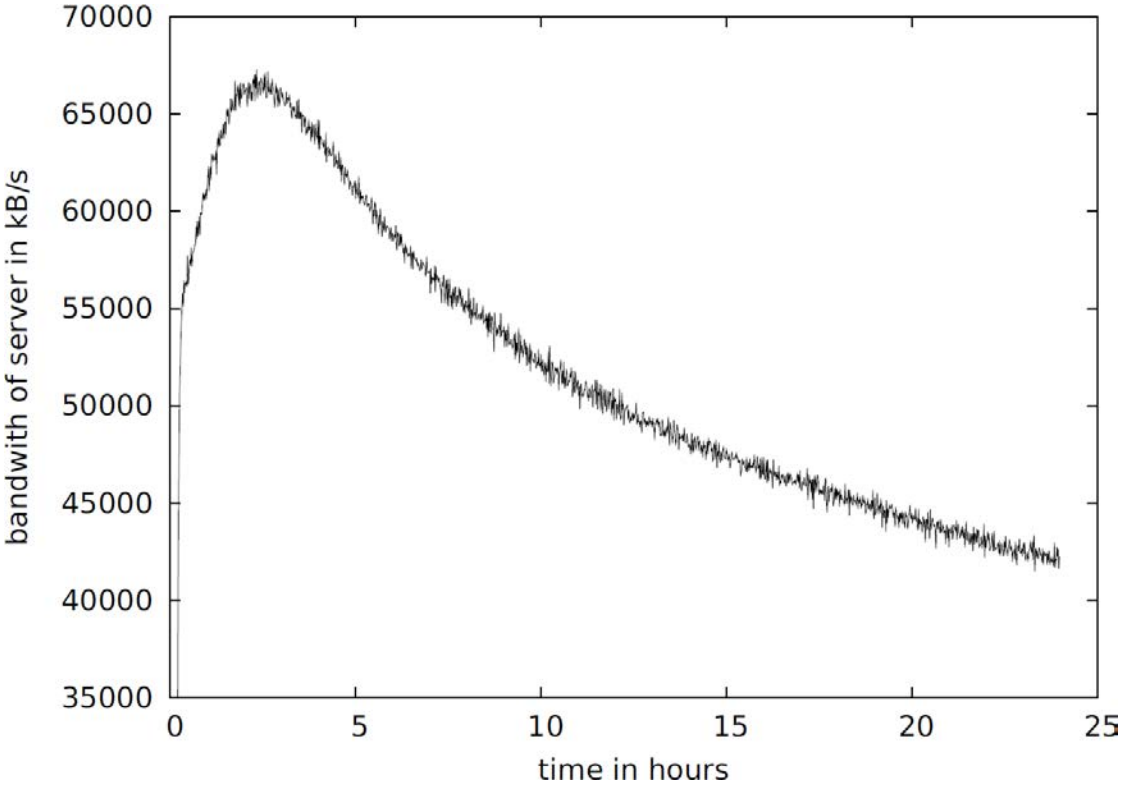


FIGURE 4.6: Server relative bandwidth usage over time

My approach is based on a reputation system: its ranking provides the ability to pick out trusted referees. I checked how much bandwidth my own in-built reputation mechanism uses in those 4KB/s. It appears that, compared to the real size of the game protocol messages, it uses only around 1.5% of all network data sent (as illustrated by Figure 4.7).

Reputation messages are sent only when needed, as I really took care about this part of the reputation system, and they also are small. They share roughly the same size as the action messages and weight 16 bytes, they contain a descriptor of a node, and the new reputation value I want to share for this node.

4.5.4 CPU load

To estimate the CPU overhead introduced by my refereeing system, I assigned 1 *" cpu load point"* to every action/state creation (cpu usage associated with the game software) and to every action/state test (cpu usage induced by my approach). Please note that this

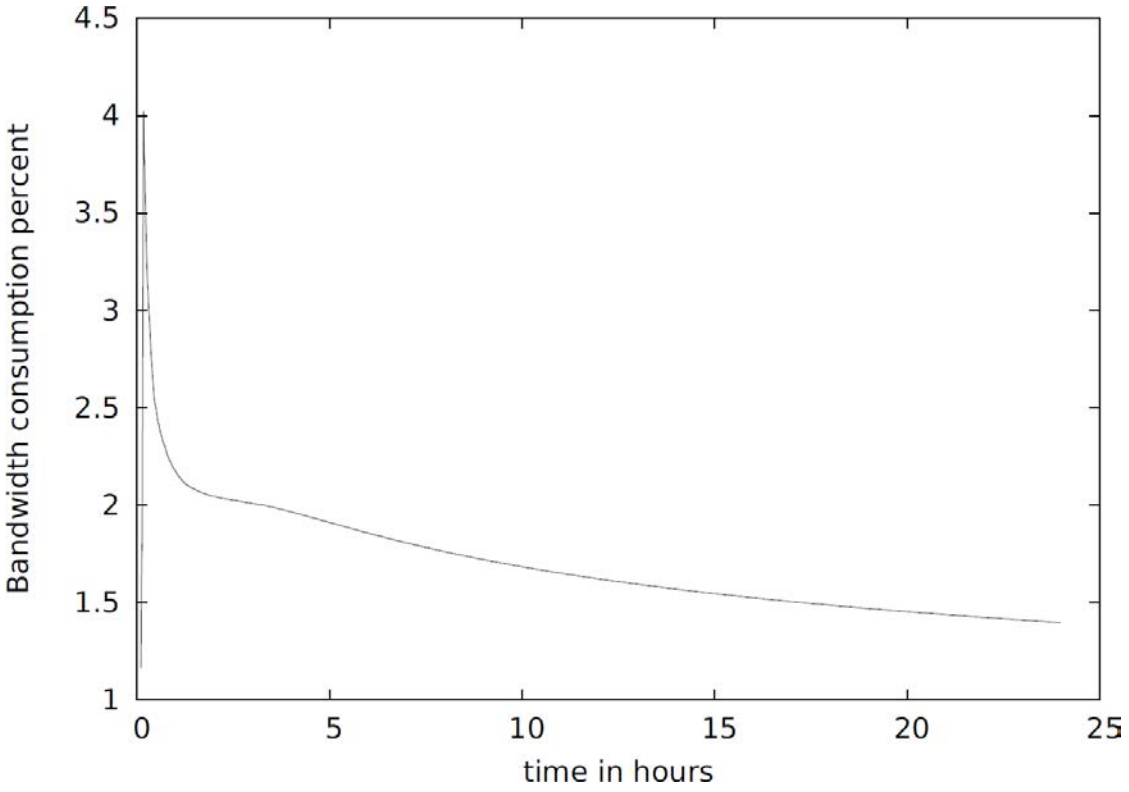


FIGURE 4.7: Bandwidth consumed by the reputation system

method inevitably leads to an overestimation of the overhead induced by my approach. In a real implementation the CPU usage associated with tests is really small compared to the other CPU dependent operations a game can produce, such as Nvidia PhysX or scenery creatures random walk animations. Referees do not have to handle such computations as they are only related to visual effects having no impact on the game engine.

Number of referees	One	Three	Five
CPU overhead	27,85%	64,59%	210,56%

TABLE 4.1: CPU overheads

I computed the values of the CPU overhead compared to running only the player code, and display the results in Table 4.1. Results are not linear as the multiplication of referees introduce an increasing amount of referee need per battle. This leads to an overall CPU increase to handle the supplementary messages sent. Obviously, my CPU overhead is closely related to the number of referees per battle. In spite of my test skipping optimization, the 5 referees per battle configuration induces a prohibitive overhead. Considering that most current games require a dual CPU core, a 100% CPU overhead on a quad- core processor seems an acceptable maximum. Once again, I wish to emphasize that my CPU load measurements are vastly overestimated.

I also monitored on desktop PCs the CPU usage of several games that are typical targets for my solution. These games use roughly 50% of the quad-cores and nearly fully the dual-cores. Multiplayer online battle arenas such as *League of Legends*, *Defense of the Ancients 2*, and *Bloodline Champions* all ran perfectly when bound to only one or two cores. MMORPGs like *Tera online* and *Rift* ran also without issues when placed on 2 cores only. The more recent *Guild Wars 2* showed some different frame rates when bound to 2 cores only, as Extreme graphics options can produce a bottleneck effect by increasing the quantity of information the CPU sends to the GPU.

4.5.5 Cheat detection ratio

With respect to cheat detection, since there are no available statistics on the number of cheating occurrences that go undetected in the gaming industry, I set the percentage and distribution of incorrect nodes arbitrarily, while trying to remain realistic with respect to both my own gaming experiences and the literature on byzantine behaviors in P2P networks.

For instance, there are diverse malicious behaviors in the context of gaming. A person using the service may want to cheat as a player, but may not want to disturb other players. Conversely some attackers only focus on damaging the system and do not care to play at all. The distinction between *player reputation* and *referee reputation* reflects this analysis: it implies that the player component and the referee component on a single node can behave independently from one another.

In the literature about reputation systems[86], 5% is often a higher proportion of incorrect/malicious nodes compared to what is commonly considered. I chose to stretch this value even further: I set the proportion of potentially malicious players to 30% and the proportion of potentially malicious referees to 10%. The latter is lower than the former because a lack of reputable referees has a perverse effect on the simulation: it decreases substantially the number of battles that can proceed, and hence the overall number of cheating attempts.

To study the impact of the proportion of incorrect referees on the quality of the cheat detection in my solution, I initially ran simulations with a proportion of incorrect referees varying between 0% and 45%. Figure 4.8 illustrates the rapid decrease in the number of battles per day that such a variation brings. I believe that 10% of incorrect referees remains a high value anyway, probably above the proportion any real game faces.

Despite this negative outcome my simulation results shown in table 4.2 demonstrate that, regardless of the number of battles that do proceed, *the proportion of undetected*

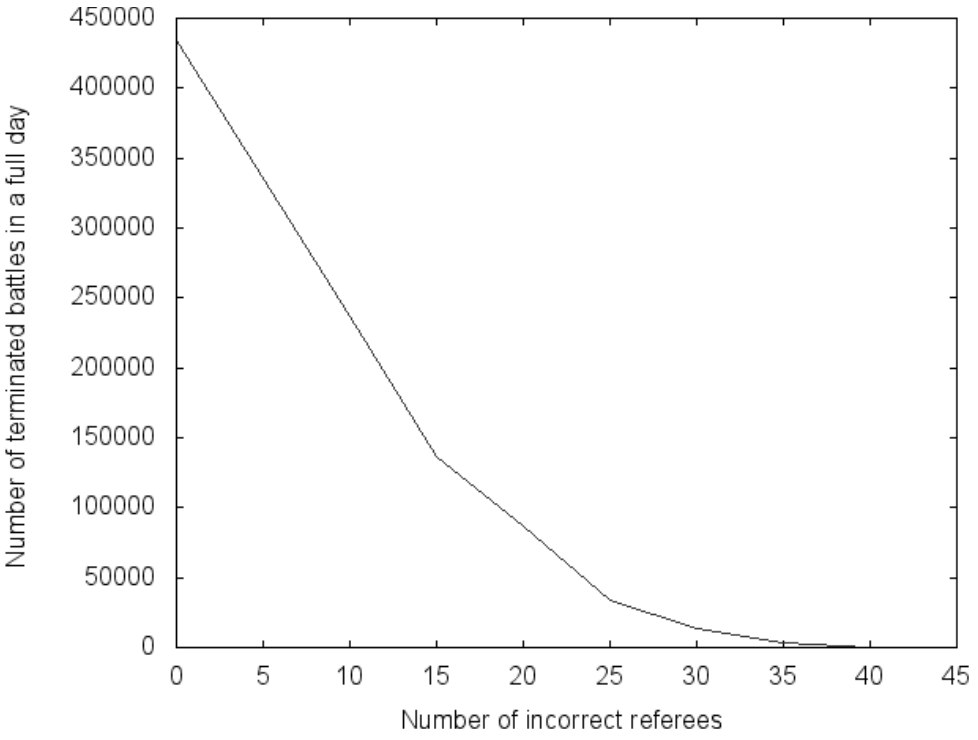


FIGURE 4.8: Number of possible battles as the proportion of correct referees decreases

Number of referees	One	Three	Five
Cheat percentage	1,17%	0,0128%	0,0099%

TABLE 4.2: Undetected cheat ratio

cheating occurences always remains stable under 0,013% with three or more referees per battle. This shows that my solution is effective for detecting and avoiding both incorrect referees and incorrect players.

In table 4.3, I correlate the measures from tables 4.2 and 4.1 to obtain the CPU overhead multiplied by the percentage of cheating attempts that went undetected. This is done in order to understand that even though the CPU overhead increases, a solution with multiple referees can be beneficial. The results show that the "many- referees" configurations are CPU cost efficient compared to the single referee one as they detect more cheat attempts. The most CPU efficient configuration is the 3 referee per battle approach.

Number of referees	One	Three	Five
CPU overhead multiplied by % cheat undetected	32,6625%	0,83%	2,10%

TABLE 4.3: CPU overhead relative to undetected cheat percentage

4.6 Performance in distributed environments

In order to assess my system in a distributed environment, I implemented a prototype of my platform in Java. Each node (both player and referee entity) is a process containing three threads: the player thread, the referee thread and the UDP network communications queues thread.

I conducted two waves of experiments. The first wave of experiments runs 150 nodes locally on a single quad 8 cores Xeon CPUs 2.4GHz. The second wave tests my implementation both on PlanetLab[90] and on Grid'5000 [91]; its aim is to double check the results of the first wave, and also to assess the ease of deployment and the scalability of my platform both in a dedicated and in a non-dedicated network. Every experimental measure is a mean value computed from 10 different runs; since the variance of every measure was extremely low, I consider that these results are reproducible and do not justify a larger number of runs.

Every experiment processes input that reproduces real in-game interactions. For this purpose, the input is a randomly generated reproduction of the distributions associated with node latencies, battle outcomes and mean times between battles coming from the League of Legends database. I arbitrarily set a proportion of malicious nodes: 30% cheat as players and 10% perform incorrect refereeing.

4.6.1 Evaluation in a dedicated environment

This set of experiments corresponds to the first wave: the platform runs 150 nodes on a single host. The resulting assessment discards the impact of the network and of other applications on the performance of my platform.

The first two metrics of my assessment are inspired from the literature about failure detectors [92]. Failure detectors have a similar objective: to discriminate between correct and incorrect nodes.

- The *average mistake rate* represents the proportion of wrong assessment of events my system makes over time (correct events judged incorrect and vice versa). I compute it by dividing this number of wrong assessments the system makes by the total number of assessments.
- The *correctness duration* gives the average amount of time elapsed on every node since their last incorrect assessment.

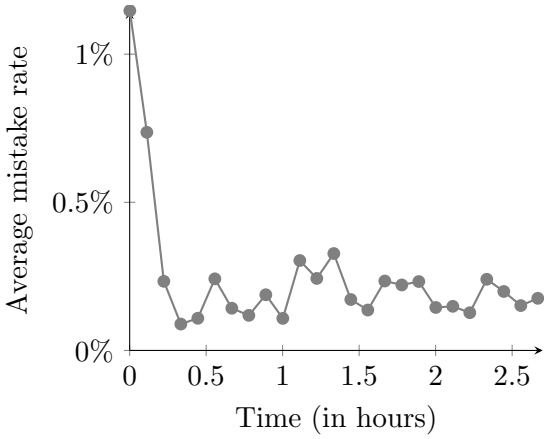


FIGURE 4.9: Average mistake rate of my platform

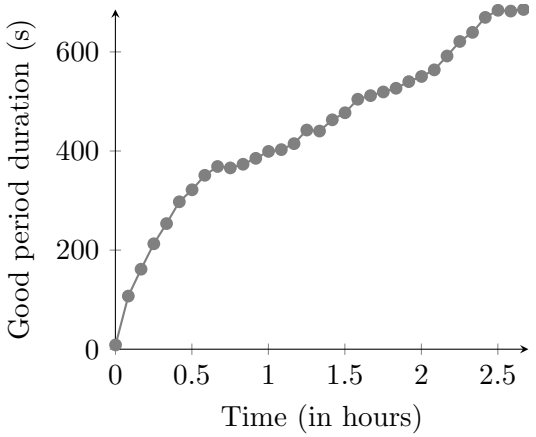


FIGURE 4.10: Correctness duration of my platform

Figure 4.9 gives the average mistake rate of my implementation: after a short initialization phase of 30 minutes (the duration of two battles), the average mistake rate remains under 0.5% and converges slowly towards 0.25%.

Figure 4.10 gives the average correctness duration of my platform. Its value increases steadily, demonstrating that the node assessment of my platform keeps getting better over time.

Although their mistake rate and their correctness duration improve over time, referees remain fallible. Please keep in mind that adding more referees would allows for an even higher cheat detection ratio.

I also measure the bandwidth and CPU consumption of my platform.

In the case of *bandwidth usage*, I isolate the consumption of the reputation system from that of the whole cheat detection platform. I measure bandwidth consumption by logging every message and its size once packed into the network, and then by adding them all.

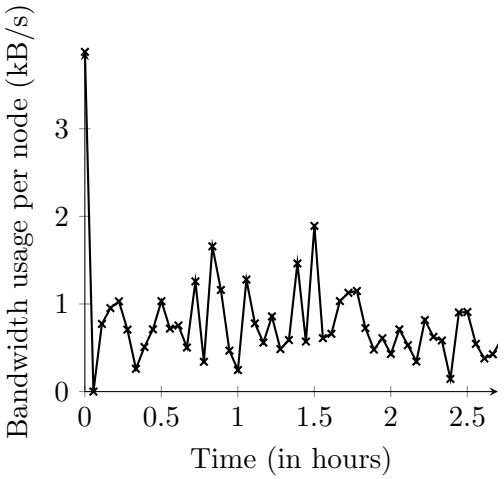


FIGURE 4.11: Bandwidth usage of my platform over time

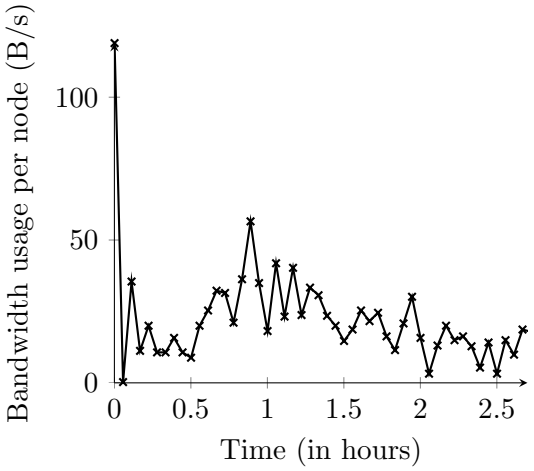


FIGURE 4.12: Bandwidth usage of my reputation system over time

Figure 4.11 shows the total bandwidth consumption of my implementation. It peaks at the launch of the system, when all nodes enter the initialization phase and exchange a lot of messages in order to identify potential referees. The bandwidth usage quickly decreases and stabilizes under 2kB/s.

Figure 4.12 isolates the bandwidth consumption of my reputation system, which is much lower than that of the whole platform: on average 2.5%. To achieve such a low consumption, my reputation system implementation delays the propagation of reputation values. Every node only sends reputation values that have changed by more than 5%. Overall the bandwidth consumption of my reputation system remains under 0.12kB/s per node through more than two hours and a half of experiment. In comparison, the bandwidth consumption of most games averages around 5kB/s [89].

The *CPU load* of a cheat detection system is also critical: a system that consumes too much CPU intrudes on the game experience. Figure 4.13 shows that the highest CPU

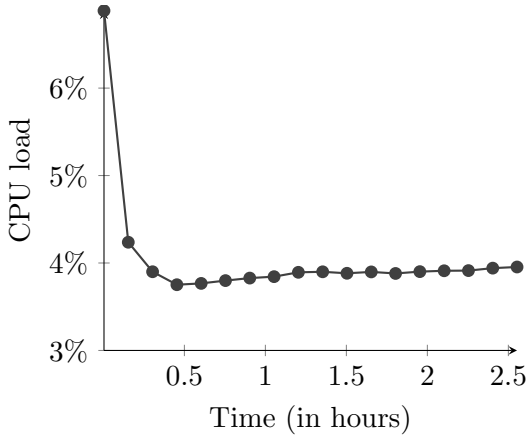


FIGURE 4.13: Average CPU load per node of my platform

Latency range (ms)	Number of nodes
0<=latency<50	69
50<=latency<100	28
100<=latency<150	33
150<=latency<200	10
200<=latency<250	10

TABLE 4.4: PlanetLab node latency distribution

load corresponds to the initialization phase where nodes assess one another with fake requests. Once my platform stabilizes, the average CPU load drops by 50% to less than 4% of the total CPU computational power of a node.

4.6.2 Deployment on Grid'5000 and on PlanetLab

I conducted a first test of scalability by deploying my platform on 1200 nodes of Grid'5000 [91]. All 1200 player nodes are located on the single cluster of Rennes and execute the same code as that of the experiment described in Subsection 4.6.1. Even though this experiment involves a large number of nodes, latencies are too low to be realistic (below 1 millisecond). Therefore, I carried out the same latency injection as the one described in Subsection 4.5.2.

To assess its behavior in a non-dedicated environment, I also deployed and ran my platform on 150 nodes from all around the world through PlanetLab [90].

Table 4.4 gives the ping distribution of my allocated PlanetLab nodes. Some of these nodes had very little bandwidth available (less than 50kB/s), yet the results that follow show the good performance of my implementation despite this limitation.

Figure 4.14 plots the average number of battles that every node manages to finish over time. The Grid5000 deployment confirms the results obtained on the fully dedicated

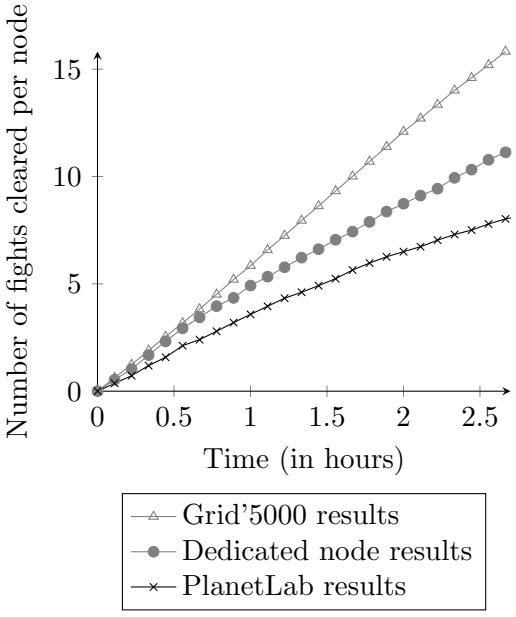


FIGURE 4.14: Comparison of the numbers of cleared fights

node, however it eventually clears 40% more fights. The reason is simple: 1200 nodes offer far more possible fights than 150. With nearly 40% of incorrect nodes, a higher number of nodes increases the availability of correct referees at any given time. With 150 nodes, the PlanetLab results shows a slowdown of 50% when network conditions are worse by at least two times in the majority of the nodes. This result shows that an overall bad network condition will impact the solution but the presence of fast replying referees still avoid a major slowdown.

Another performance metric is the average amount of time a player must wait until it acquires enough suitable referees to start a fight. Figure 4.15 compares the waiting times in all three environments and the outcome is similar to that of the previous comparison. The deployment of 1200 nodes on a Grid5000 cluster performs much better than the deployment of 150 nodes on PlanetLab, and the results of the dedicated node are a little better but remain pretty close to those of the PlanetLab deployment.

4.7 Conclusion

MMOGs built on top of fully decentralized P2P architectures may scale, but prohibit the implementation of a refereeing authority that is totally trustworthy.

Several research works [22, 48, 93, 94] propose distributed cheat detection mechanisms. In [22] and [93] the authors use trusted entities to handle security, and in [48] super-peers serve as proxies for overall security. These solutions introduce the basic mechanisms

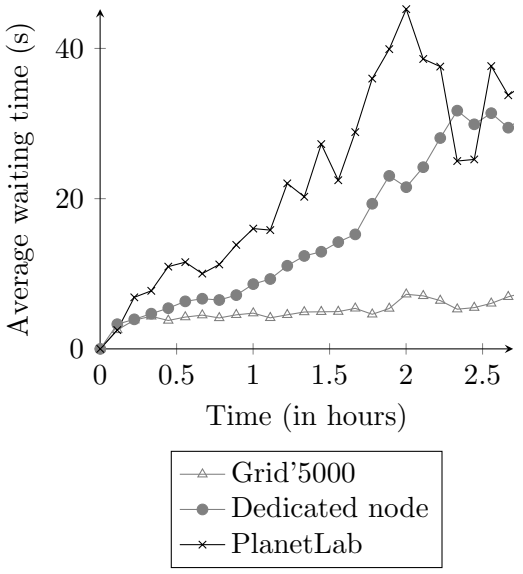


FIGURE 4.15: Comparison of the cumulative waiting times between fights

for fully decentralized P2P gaming. However they do not tolerate a high number of incorrect nodes. RACS [93] limits the arbitration to a single referee. It deals neither with cooperations amongst multiple referees, nor with the selection of trusted referees.

In this work, I propose a solution based on a reputation system. It tests the behavior of nodes by submitting fake refereeing requests and then picks out nodes it identifies as trustworthy to referee real game actions. To improve the detection of dishonest arbitrations by malicious referees, every request can be submitted to multiple referees concurrently.

I ran experiments to test whether my solution is viable real network conditions are involved. Experimental results included in this work show that it is possible to provide an efficient anti-cheat mechanism in peer to peer MMOGs without degrading their scalability.

My approach could be extended to detect other kinds of cheating, such as malicious behaviors that remain legal within the game. For instance, *goldfarming* consists in gathering virtual resources in order to sell them for profit in the real world; it is a source of intense frustration for MMORPG players because it tips the balance in favor of wealthy users. Using the multi- agent paradigm might help solve this issue. Agent-based systems are good at representing and monitoring complex behaviors and rich interactions between nodes. One such system could log player actions and referee decisions to spot abnormal behaviors linked to stolen accounts and gold farming. Additionally this improvement could allow retroactive removal and replacement of bad referees.

Chapter 5

Using reputation systems as generic failure detectors

Contents

This chapter is not a work in relation with gaming but shows the usability of a reputation system in a generic service: failure detection. Failure detection plays a central role in the engineering of such systems. Chandra and Toueg introduced in [95] the notion of unreliable failure detector (FD). An FD is an oracle which provides information about process crashes. It is unreliable as it can make some mistakes for a while; for instance, some live nodes can be considered as having crashed. FDs are used in a wide variety of settings, such as network communication and group membership protocols, computer cluster management and distributed storage systems. Numerous implementations of FDs have been proposed, where each node monitors the state of the others. However, most FD implementations have two severe limitations:

- they consider all the nodes in a same way, there is no distinction between well and bad behaved nodes;
- local oracles gather information from the other nodes without any coordination [96–98].

In stable and homogeneous configurations such as clusters, where nodes of a same type are linked through low latency networks and subject to crash failures at the same rate, these limitations have a low impact on the quality of the failure detection. However, in large and dynamic systems such as gaming platforms or large cloud infrastructures, nodes are very different: some nodes (eg. Server) are powerful and connected to the network with a high speed link whereas some others have a limited power and slow connections. Taking into account such differences is essential for the quality of the detection. Furthermore, in such dynamic environments sharing information on the state of the nodes could greatly increase the global view of the distributed system. If one node has a good connection to the other ones, it can share its view to slowly connected nodes and thus prevent wrong views about failures.

In this chapter, I propose a new collaborative failure detector which exploits reputation of nodes to increase its detection quality both in terms of detection time (completeness) and mistake avoidance (accuracy). To obtain reputation information nodes periodically exchange heartbeat messages. The reputation of a node dynamically increases if it sends its heartbeat on time, and decreases if some heartbeats get lost or arrive after the expected dates.

I conducted an extensive evaluation of my failure detection on distributed configurations using real traces to inject failures and message losses. I show that my detector outperforms well-known implementations [97, 99]: it provides a better accuracy while keeping short detection times, especially when the network is subject to message losses.

The rest of the chapter is organized as follows. Section 5.1 presents the reputation system I use to implement my failure detection service and details the detector implementation. Section 5.2 describes two standard failure detector implementations I then compare to my solution in the performance evaluation of Section 5.3. Finally, Section 5.4 explores related work and Section 5.5 concludes the chapter.

5.1 Detecting failures with a reputation system

My solution uses a distributed reputation system to detect failures. A reputation system [59] aims to collect and compute feedback about node behaviors. Feedback is subjective and obtained from past interactions between nodes, yet gathering feedback about all the interactions associated with one node produces a rather accurate representation of its behavior. In my case, the reputation system focuses on behaviors that fall within the scope of a given failure model.

The reputation system I present in this section is basic and aims to reproduce the qualities of a good reputation system according to [60]: fast convergence, precise notation of nodes, resistance to malicious nodes, small overhead, scalability, and adaptivity to peer dynamics. My reputation system can be used for a wide variety of middlewares and services. In a previous work, I describe in details and use this reputation system to support another kind of application, namely cheat detection in massively multiplayer online games [83].

5.1.1 Assessment of the reputation

Every node stores a local estimation of the *reputation* associated with every node in the network. In my system, a reputation value belongs to $[0, 1000[$. Value 0 represents a node which never delivers its service correctly, whereas the reputation value of a very trustworthy node tends to 1000. Initially, a node with no known history in the network has its *reputation* value set to 0.

A reputation assessment primarily consists in comparing inputs from neighborhood nodes with an expected behavior. Applying this scheme to failure detection is simple: if a node sends its heartbeat in a timely manner its reputation value increases,

otherwise it decreases according to a reputation *punishment*. I call this primary assessment a *direct* assessment; it is in all ways similar to traditional failure detection. To improve the detection in dynamic environments, I combine the *direct* assessment with an *indirect* assessment. Please refer to Chapter 4 in order to get a full description of the formulas used in our reputation system.

5.1.2 Parameters associated with my reputation system

Several parameters associated with my reputation system allow to adapt it to the requirements of the application. Fine tuning the values of these parameters requires a test phase on the target network. Subsection 5.3.1 includes a description of my benchmarking methodology to adjust the settings of my reputation-based failure detector. The present Subsection gives general pointers for the parameterization of the reputation system dedicated to failure detection.

The first obvious set of parameters α, β, γ characterizes every reputation assessment. A high value for α will confer a greater importance to past reputation values stored locally. This is useful in systems where close neighbors behave erratically. Parameter β focuses on the history of the behavior of a node. Systems with a high value for β slow down reputation decreases induced by sporadic changes in the network such as bursts of message losses. Finally, parameter γ reflects the direct impact of a new opinion on the local assessment. A high value for γ implies that the local reputation value of a node will be more sensitive to new values expressed locally or by other nodes. In the context of fault detection, I believe the system should focus on its tolerance to jitter and to message losses. As the main component of the formula for this purpose, β ought to be set to a high value.

In order to detect failures the threshold T is used again. As this threshold is a crucial metric for determining faulty nodes, I also describe how I fix its value in Subsection 5.3.1.

There are other, more secondary parameters with respect to detection. Values υ (up) and δ (down) correspond respectively to rewards and punishments. Also, a *decay* factor affects all locally stored reputation values upon a *reassessment* timeout. This strategy aims to force nodes to reassess reputations for nodes with which they have no direct interaction for some time. High values for the reassessment frequency and the decay factor reduce the detection time, but increase the rate of false detections.

```
1  Task T1 [HeartBeat / Reputation propagation]
2  Every Δ_H
3
4  |  For each known node n Do
5  |  |  Insert REPUTATION(R_n, n) in heartbeat
6  |  Send HEARTBEAT to each neighbor
7
8  Task T2 [Heartbeat reception]
9  Upon reception of HEARTBEAT from q
10
11 |  R_q ← PID(υ, q)
12 |  If HCounter = HC Then
13 |  |  { Refreshment of reputation of all known nodes }
14 |  |  For each known node n Do
15 |  |  |  R_n ← R_n − decay
16 |  |  Hcounter ← 0
17 |  For each REPUTATION(R_n, n) in heartbeat Do
18 |  |  R_n ← PID(R_n, n)
19 |  HCounter ← HCounter + 1
20
21 Task T3 [Punishment]
22 Every Δ_H * 2
23
24 |  For each known node n Do
25 |  |  If !received(HEARTBEAT, n) Then
26 |  |  |  R_n ← PID(δ, n)
27
28 Task T4 [Decision]
29 Upon asking a detection for node n
30
31 |  If R_n > T Then
32 |  |  return NodeCorrect
33 |  Else
34 |  |  return NodeFaulty
35
```

Algorithm 1: Reputation based failure detector algorithm

5.1.3 The reputation based failure detector

Algorithm 1 presents my reputation-based failure detector implementation. In task T1, every node sends its neighbors a heartbeat message every Δ_H. Every heartbeat encapsulates the reputation information the node cares to propagate. When a node p receives a heartbeat message from a node q (task T2), it computes a new reputation $R_q(t+1)$ which rewards q with a maximum update value $a = \upsilon$ (line 11). Then for each reputation information included in the message, p updates the reputation of the corresponding node (lines 17–18). For an eventual decrease of the reputation of all known faulty nodes, a reputation reassessment occurs every HC heartbeats: p then applies a *decay* factor to all locally known reputation values (lines 14–15). To boost local detection, p punishes nodes which haven't sent heartbeats for two periods (task T3). I used this value to be responsive whilst being resistant to small jitters. Finally, task

T4 handles fault detection requests from the application layer. My detector considers a node is correct if its reputation value is greater than a given threshold T.

5.1.4 Scalability

In order to scale up to a large number of nodes, my reputation system imposes a network topology that avoids all-to-all communication. We conceived a simple algorithm (Algorithm 2) for the random generation of connected digraphs where every *common node* has an average degree of 3. In order to reproduce the skewness of connectivity in large scale overlays, my algorithm also randomly designates *super nodes* that possess direct links towards a third of the network. A node has a probability P_{super} of being a *super node*. My algorithm uses a coordinator node that builds a static network topology for all connected nodes. Initially each node sends its identifier to the coordinator (Task T1). The coordinator then sends each node the list of its neighbors in the overlay (Task 3). In the algorithm, $Random()$ generates a random number in the interval $[0, 1]$ and $Subset(S, c)$ returns a random subset of set S with a cardinality equal to c.

```
1  P_super ← 0.1
2  min_neighbors ← 2
3  max_neighbors ← network_size/3
4  network ← ∅
5
6  Task T1 [Node Initialisation]
7
8  | Send ID(p_i) to Coordinator
9
10 Task T2 [Neighbors reception]
11 Upon reception of NEIGHBORS(list) from Coordinator
12
13 | neighbors ← list
14
15 Task T3 [ID reception]
16 Upon reception of ID(p_j)
17
18 | network ← network ∪ p_j
19 | If |network| = network_size Then
20 | | For each node n in network Do
21 | | | If Random() > P_super Then
22 | | | | list ← Subset(network, max_neighbors)
23 | | | Else
24 | | | | list ← Subset(network, min_neighbors)
25 | | | Send NEIGHBORS(list) to n
26
```

Algorithm 2: Overlay generation

As I showed in [83], my reputation system scales extremely well with respect to the number of nodes involved. Even though the solution may sound resource intensive, it only

uses 160 bytes per second per node in a system with 30000 nodes running the reputation system. This is due to the small size of the reputation data that gets exchanged.

In the performance evaluation of Section 5.3, I limited the size of the network to 10 nodes to allow for a large number of experiments over varying configurations. In particular, Subsection 5.3.8 studies and compares the bandwidth consumption of the three approaches in various experimental scenarios. However we also ran an experiment involving 250 nodes to check how my reputation-based failure detector behaves over a larger network. I observed that, regardless of the network size, the bandwidth consumption of my detector remains steady. Every node that participates to my reputation system consumes an average bandwidth of 144 Bytes per second, with almost no deviation. I consider that this value is very low and well within the capacity of most Internet connections nowadays.

5.2 Comparison with other failure detectors

In order to assess my reputation oriented approach, I compare it with two other oracle-based failure detectors: Bertier [97] and Swim [99]. These failure detectors constitute references of efficient and well-known failure detectors. They are briefly presented in this section, along with a first assessment of each detector's impact on the network.

5.2.1 Bertier's failure detector

Bertier's failure detector combines one of Chen's estimations [100] for the arrival time of heartbeat messages and a dynamic safety margin based on Jacobson's algorithm [101].

Chen's estimation computes the expected arrival time EA of heartbeat messages, and adds a constant safety margin α to avoid false detections caused by transmission delay or processor overload. EA results from adjusting the theoretical arrival time of the next heartbeat with the average jitter incurred upon the n latest heartbeat receptions. The value of α requires a calculation with respect to QoS requirements prior to starting the system; it can not account for radical alterations of the network behaviour.

Bertier's failure detector solves this issue by using Jacobson's algorithm to compute a dynamic safety margin. Jacobson's estimation assumes little knowledge about the system model; it is used in TCP to estimate the delay after which the transceiver retransmits its last message. This estimation relies on the error incurred upon reception with respect to the estimated arrival time, and includes user-defined parameters to weight the result.

5.2.2 Swim's failure detector

Swim [99] relies on a ping approach. An initiator node invites k other nodes to form a group, pings them and waits for their replies. If a node does not reply in time, the initiator then judges this node as suspicious and asks the other group members to check the potentially faulty node. If this node remains silent after three consecutive pings from all group members, the detection is confirmed and the node is finally considered as incorrect.

Swim's failure detector allows a fair comparison in that, instead of requiring an *all to all* communication, it shares the same type of network footprint as a reputation system.

5.2.3 Communication complexity

The three algorithms I compare have different impacts on the network.

Bertier. Bertier's failure detector induces all to all communication. Upon every period, every node sends a heartbeat to every other node: their message complexity is n^2 messages per period.

Swim. The failure detector in Swim has two operational modes: a standard one where the initiator assesses its group of K nodes, and a degraded mode where all K nodes assess a non-responsive node. This derives two message complexities: (i) $k * n$ messages per period in standard mode, and (ii) $k * n^2$ in the worst case of the degraded mode.

My reputation-based detector. Reputation systems usually incur a high communication complexity, so I reduced my network footprint as much as possible. Nodes send periodic heartbeats to their neighbors only, and propagate reputation data by piggybacking it on the heartbeats. To reduce network consumption even further, nodes only emit reputation values that differ significantly from the last emitted value. This produces a difference of behavior between good and bad nodes as I will save a lot of bandwidth on good nodes reputation propagation. In my experiments, setting the minimal variation before emission to 50 on a $0 - 5000$ scale reduced message size by 50%, with no effect whatsoever on the quality of the detection.

Let d be the average degree of nodes in the system: the complexity of my detector is $d * n$ messages per period.

5.3 Performance evaluation

This section presents an evaluation of my approach. I assess its performance and compare it with both Bertier's [97] and Swim's [99] state of the art failure detectors. Throughout this section I use *BertierFD*, *SwimFD*, and *RepFD* to refer respectively to Bertier's, Swim's, and my reputation-based failure detector.

5.3.1 Experimental settings

Application settings All scenarios are run on ten nodes for a duration of five minutes, with a heartbeat/ping period of one second. Each experiment is run fifty times. The standard deviation consistently remained very low: under 4% for all my experiments. Hence I chose not to include it in my figures.

Network settings I ran my experiments on a cluster made out of dual-Intel Xeon X5690 running at 3.47Ghz and equipped with 143GB of RAM. Since my cluster incurs near zero latency, I injected a 59ms latency for each message to reproduce typical user broadband connections. This value comes from my study of user experience that capitalizes on statistics published by a very popular online game [102]. My code uses the UDP protocol for message exchanges, thus preventing detections caused by connection closures.

BertierFD relies on all to all communication so I organized the network in a clique. For SwimFD, I fixed the membership set to involve all ten nodes. Finally, I generated the topology for my reputation system at random upon every experiment by using the algorithm described in Subsection 5.1.4. The resulting graph is connected and the degree of every node is greater than or equal to two. This topology limits communications overhead while ensuring the liveness of the propagation of reputation information among nodes that generate the global view.

Failure detector settings I set $\Delta_H = 1s$ for all the assessed failure detectors. BertierFD and RepFD send heartbeat messages every second. SwimFD also sends ping messages every second in order to obtain coherent detection times.

BertierFD requires an additional setting: the initial detection time beyond which a missing heartbeat determines its sender as suspicious. I followed the original implementation and set this value to Δ_H. All other parameters were also set to the values advocated in [97].

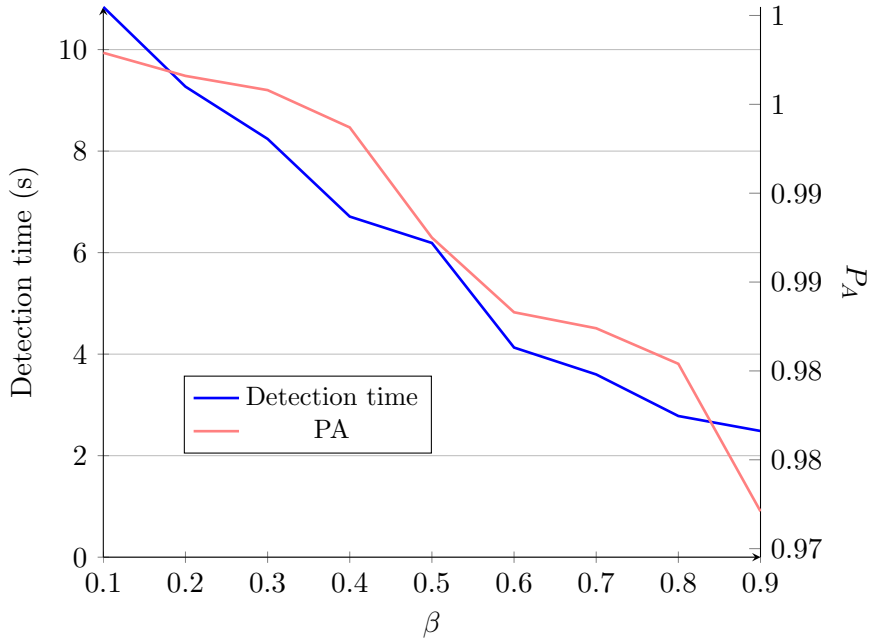


FIGURE 5.1: Impact of beta and gamma on the quality of the detection

RepFD relies on a reputation assessment which requires some parameterization. The essential parameters of this assessment are the weights α, β, and γ of the PID formula that computes reputation values, as well as the threshold T that distinguishes suspicious nodes from correct ones according to their computed reputation value. To adjust the values for these parameters, we first ran brute force simulations on top of the Peersim simulator [80]. I set the reward value υ to 1000 (maximum reputation value), the punishment value δ to 0 (minimal reputation value), *decay* value to 50, HC to the size of the network, and then ran simulations where α, β, and γ varied from 0 to 1 and T varied from 400 to 800. My detector achieved its best quality of detection with the following settings: $\alpha = 0$, $\beta = 0.8$, $\gamma = 0.2$, and $T = 700$.

To reach a better understanding of why these settings work well, I experimented further on top of my ten-node cluster. I set threshold T to 700 and measured both the accuracy and the detection time of my detector with varying values for α, β, and γ. I started with a separate study of each weight. To do so, I initially incremented the value of a single weight by 0.1 between 0 and 1, and set the other two weights equal to $\frac{1-(studied\_parameter)}{2}$. Conversely to β and γ in these experiments, I observed that no value of α leads to a good quality of detection. $\alpha > 0.25$ prevents any detection, while values below 0.25 induce detection times of at least 3.74 seconds. I will show in the following experiments that this is a poor detection time, but one can reach this conclusion intuitively as it represents almost four times the period between heartbeats. Hence the first conclusion I can draw from my study is that the local view of past detections bears too much influence on future detections and should be discarded entirely.

I therefore focused my study on β and γ: I incremented the value of γ by 0.1 between 0 and 1, and set $\beta = 1 - \gamma$. Figure 5.1 plots the detection time and the accuracy for all values of β in this last series of experiments. Introduced in [100], query accuracy probability (noted P_A) reflects the probability that a failure detector's output is correct at any random time. These results confirm those reached through my simulations: beyond $\beta = 0.8$ the detection time will not decrease much further, but the accuracy starts shooting down. My understanding is that detection assessments from distant nodes, included in γ, actually slow down the local detection and help avoid mistakes. Meanwhile, β has a positive influence on detection time as it reflects the consistency between the latest local detections and those of neighbor nodes.

This extensive study leads to the following settings for my reputation-based detector throughout my experiments: $\alpha = 0$, $\beta = 0.8$, $\gamma = 0.2$, and $T = 700$.

5.3.2 Measuring the accuracy of multiple detectors

In order to measure the accuracy of the detection, I introduce a metric we call *global node correctness*. The *global node correctness* of a node n is the average number of correct nodes that deemed n correct. A value of 1.0 means that all nodes consider n as correct.

First I define a *correctness* value for any node n of the network as:

$$correctness(n, p) = \begin{cases} 1, & \text{if } n \text{ is deemed correct by p} \\ 0, & \text{otherwise} \end{cases}$$

With my reputation system, a node n with reputation R_n is deemed correct if $R_n > T$.

To calculate *global node correctness* at any given moment, I compute the *global node correctness* of a node n as the average correctness associated with n throughout the network.

$$global\ node\ correctness(n) = average(\ correctness(n, p), \forall\ p \in network)$$

For example with a perfect detector all nodes will detect incorrect nodes as incorrect, therefore the average *global node correctness* of the incorrect nodes will be 0.

5.3.3 False positives in the absence of failures

I first measure and compare the accuracy of the studied failure detectors by testing for false positives in the absence of failures.

BertierFD initializes its heartbeat detection period to Δ_H, and then requires some time to estimate the real RTT of nodes which includes the latency. It takes approximately 14 seconds to converge but behaves extremely well once the system stabilizes : it does not produce a single false positive. The resulting plot, a straight line, is therefore uninteresting and I decided not to include the figure in this work.

SwimFD also converges very fast: its stabilization time is similar to that of BertierFD, around 14 seconds. However, even in the absence of failures, SwimFD does not handle jitter of connection very well. As shown in Figure 5.2, small changes of latency quickly drive nodes into SwimFD's list of suspicion. A suspicion does not directly translate to a detection for SwimFD, though: an overdue heartbeat launches the degraded mode where nodes in the same group exchange their views on the suspected nodes. Overall SwimFD introduces a lot of false suspicions and thus generates significant network usage overhead, but it does not produce false detections.

RepFD is noticeably slower at converging: it takes about 30 seconds to stabilize. This is the time required by the underlying reputation system to build its view of the network. Upon stabilization, RepFD also behaves well: it produces no false detection in the absence of failures. Figure 5.3 plots the average reputation value over time. RepFD initializes reputation values at 0, but starts storing them after the first call to *PID* function: hence the first reputation value represented on the graph is higher than 600. Since a correct behavior systematically entails a reward v for the observed node, the reputation of correct nodes quickly converges towards the highest possible value and will never decrease below the 700 threshold value.

5.3.4 Permanent crash failures

After analyzing the behavior of the detectors in a failure free environment, we introduce silent crash failures. At the initialization of the experiment, the system randomly designates two nodes as faulty. Nodes programmed for incorrect behaviors crash silently and never recover. An incorrect node implements the silent crash by shutting down its network connections after the full stabilization time of the studied detectors. I therefor set the crash time to 50 seconds in order to ensure that all systems have fully converged.

The detection time corresponds to the time elapsed between the moment the failure occurs and the moment all correct nodes detect the failure. As such it is relative to the length of the heartbeat period but, in my case, setting the same heartbeat period for all three detectors offers a fair comparison.

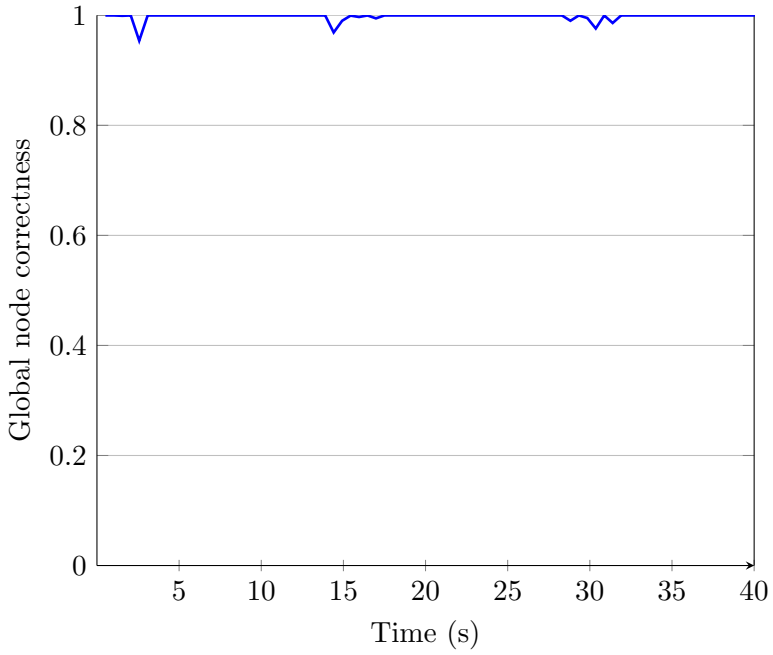


FIGURE 5.2: Accuracy of SwimFD in a failure free environment

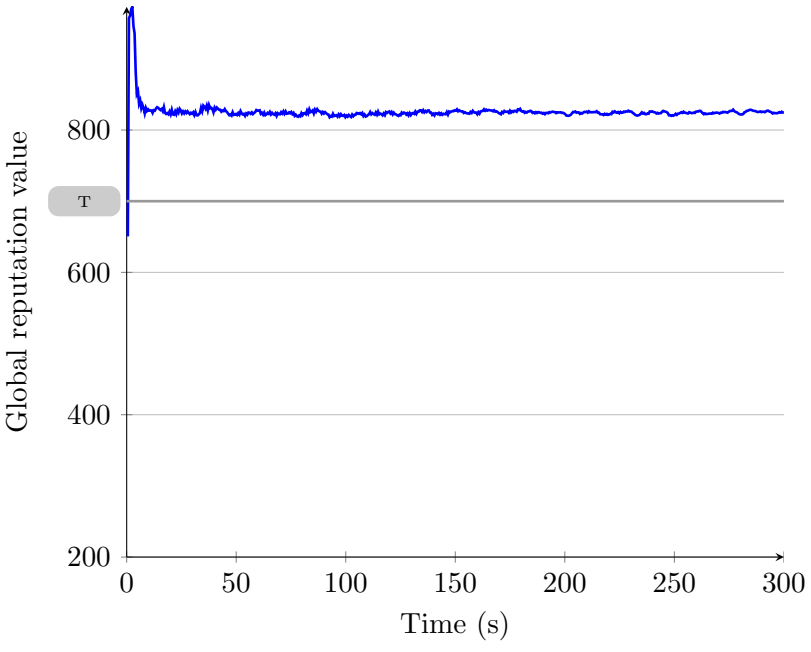


FIGURE 5.3: Accuracy of RepFD in a failure free environment

Table 5.4 displays the average detection times exhibited by the detectors. All three detectors react within the same timeframe. SwimFD may seem faster: its displayed detection time is roughly 20% shorter than those of BertierFD and RepFD. However, SwimFD's detection time corresponds to the time elapsed from the moment of the crash to the moment all correct nodes have pushed the faulty node in their list of suspicion. The true detection requires three more RTTs among the nodes of the detection group.

	BertierFD	SwimFD	RepFD
Detection time (s)	2.48	1.97	2.578

FIGURE 5.4: Detection time of a permanent crash failures

I also study the accuracy of the detection for each detector. For this purpose, I measure the *global node correctness* of correct nodes and that of faulty nodes separately over time. Figures 5.6, 5.7, and 5.8 represent my results for BertierFD, SwimFD, and RepFD respectively. All three detectors behave similarly towards faulty nodes: once they start suspecting a permanent failure, they will not reconsider. SwimFD does not handle correct nodes as well as BertierFD and RepFD, however: it keeps suspecting them intermittently.

The overall conclusion of this set of experiments is that, even though SwimFD takes a little less time to reach a decision towards suspicion, it pays a heavy price in terms of accuracy. RepFD and BertierFD are much more stable in their estimation of correctness. It is important to point out that BertierFD achieves this over an all to all communication protocol, whereas a connected graph suffices for my failure detector.

5.3.5 Crash/recovery failures

The following set of experiments evaluates the behavior of the studied failure detectors in a more complex environment, where faulty nodes stop receiving/sending messages for a given amount of time and then resume communications. In my crash/recovery scenario, faulty nodes cease their network interactions every 30 seconds for a duration of 30 seconds. Please note that this scenario is not common in the literature about failure detectors; most studies focus on permanent crashes. However, crash/recovery failures are far more representative of the large and dynamic systems I target.

I first study the detection time in this series of experiments, and collect measures taken after each detector's respective stabilization time. Table 5.5 displays the average detection times for all three detectors.

My first observation is that, in this scenario too, all three detectors spot failures within the same timeframe. Yet upon closer inspection, it appears that introducing recovery

	BertierFD	SwimFD	RepFD
Detection time (s)	3.19	2.6020	2.5776

FIGURE 5.5: Average detection times in crash/recovery environments

does affect the detection times of BertierFD and of SwimFD: compared to the permanent crash scenario, they increase by 28% and by 32% respectively. Conversely, the introduction of recoveries does not impact the average detection time of RepFD. The increase is not surprising in the case of SwimFD. SwimFD does not rely on a history of past detections but on a group decision. Given its lower accuracy and the fact that it generates a considerable amount of network overhead upon suspicion, its average detection time converges towards that of the other detectors when repeated occurrences of multiple crash/recoveries complicate the detection. The more complicated it is to reach a common decision, the more time it takes for SwimFD detectors to reach it. BertierFD and RepFD both rely on past detections, so in light of the permanent crash experiment results we were expecting their detection times to be similar again. However I was underestimating the sensitiveness of Bertier's RTT estimation to latency variations. An analysis of my logs showed indeed that my cooperative reputation assessment adjusts much faster to crash/recovery than Bertier's RTT estimation computed single-handedly on every node.

I then study the accuracy of each studied detector in my crash/recovery scenario. As expected, SwimFD has a hard time avoiding mistakes in its assessment of both correct and faulty nodes. Figure 5.10 illustrates this situation: both graphs show that all nodes incur a lot of suspicion, even nodes that behave correctly throughout the experiment. BertierFD is far more stable (Figure 5.9): there are no haphazard mistakes once a node is considered either correct or suspect. But changes of state of faulty nodes lead to some degree of hesitation for BertierFD: the points where faulty nodes stop/resume their communications do not appear as sharp angles on the associated graph. This hesitation comes from the RTT reevaluation every time a change of state occurs. My reputation-based detector produces the results shown in Figure 5.11b: no jitter and clear-cut angles. This shows that RepFD responds very well to the crash/recovery scenario. Figure 5.11a explains why by plotting the evolution of the reputation values used to determine correctness. The uncertainty associated with the assessment translates to the oscillation of the reputation values, while the threshold enforces both mistake avoidance and a quick discrimination between correct and faulty nodes.

Another interesting point of my approach is that, while detectors gather reputation values, they also build a global view of the network. Therefore, even temporarily isolated nodes will still have a global view of the system while the network link is down. This provides meaningful data for identifying points of failure.

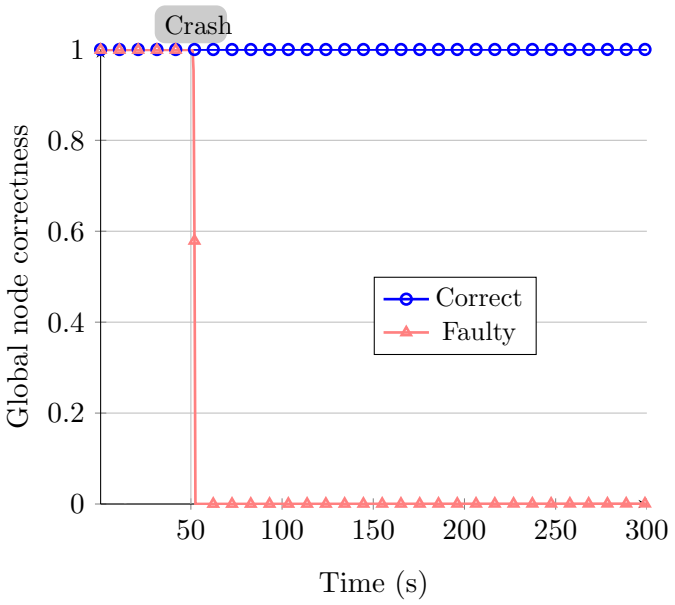


FIGURE 5.6: Accuracy of BertierFD with respect to fail silent failures

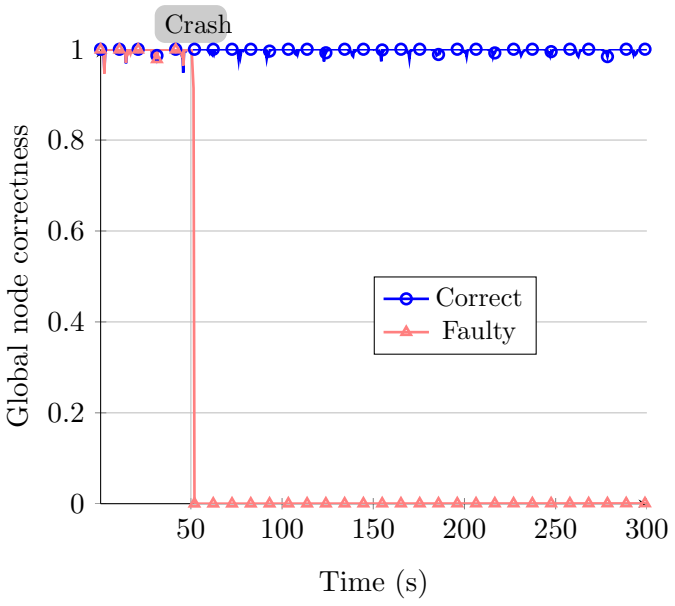
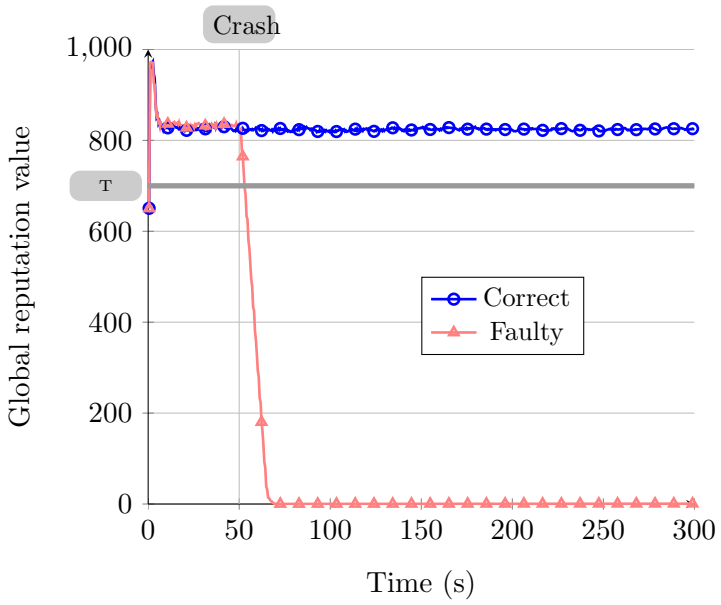


FIGURE 5.7: Accuracy of SwimFD with respect to fail silent failures

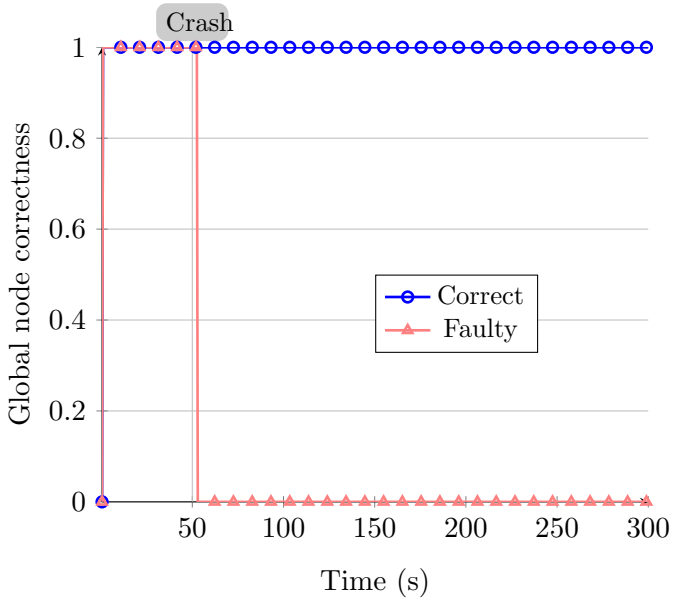
5.3.6 Overnet experiment: measuring up to a realistic trace

In order to test the failure detectors in an environment that reproduces realistic failures, I took the *Overnet* trace from the *Failure Trace Archive* [21]. *Overnet* is representative of the large and dynamic environments I target: it constantly sustains a high number of omissions from all nodes.

During each experiment, every node reads a separate node trace selected at random, and then connects to or disconnects from the system according to the trace. This creates an



(A) Reputation value of incorrect nodes



(B) Correctness with my reputation based failure detector

FIGURE 5.8: Accuracy of RepFD with respect to fail silent failures

environment where incorrect nodes omit heartbeat emissions. The average time between two disconnections is 4 seconds; incorrect nodes generate an average of 1 failure every 10 heartbeats. My testbed produced more than 280,000 node reconnections throughout this series of experiments.

I start by studying the detection time of each detector in this scenario. The presentation of my results differs from previous Subsections because average values would mask the important detection time variations incurred by the detectors. Figure 5.12 uses boxes to represent the second and third quartile; the crossing line inside the boxes represents

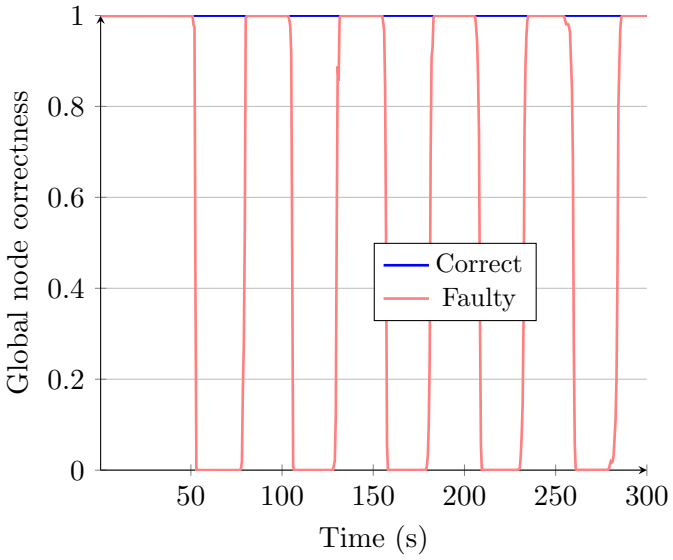


FIGURE 5.9: Accuracy of BertierFD in my crash/recovery scenario

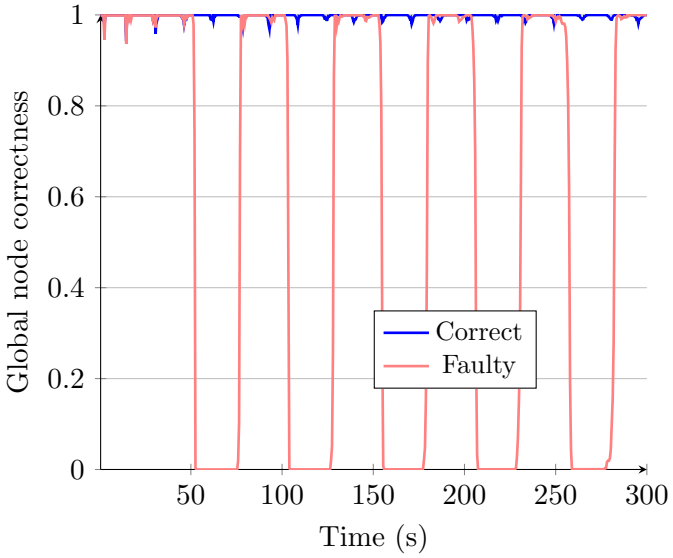
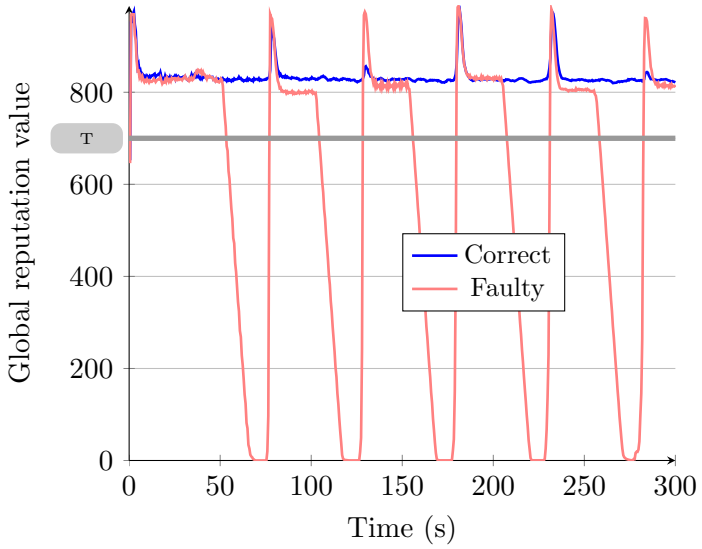
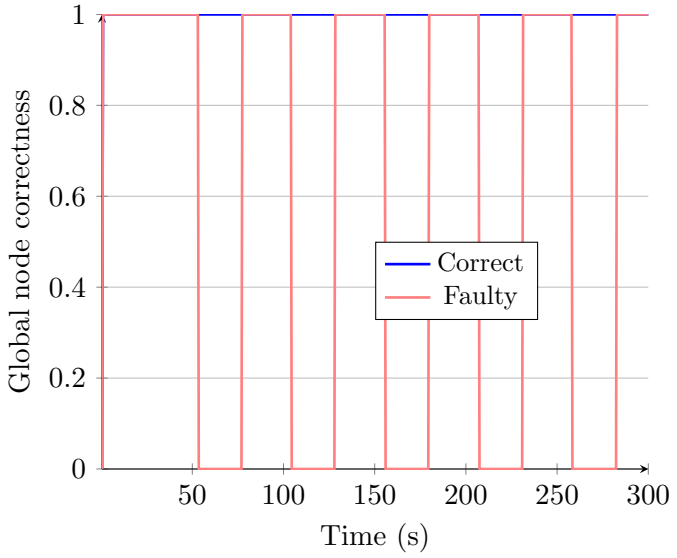


FIGURE 5.10: Accuracy of SwimFD in my crash/recovery scenario

the median; and finally the whiskers represent the 90th and 10th percentile. All three detectors produce very similar behaviors in terms of detection time: this confirms the results we obtained for SwimFD and RepFD in the crash/recovery scenario of Subsection 5.3.5. The high frequency of the reconnections explains that the detection times of BertierFD remain close to those of SwimFD and RepFD. As soon as BertierFD starts suspecting a faulty node, the latter resumes its communications and BertierFD cancels its RTT reevaluation. While this helps BertierFD maintain reasonable detection times, it seriously impacts its accuracy.



(A) Reputation values in my crash/recovery scenario



(B) Correctness deduced from the reputation assessment

FIGURE 5.11: Accuracy of RepFD in my crash/recovery scenario

5.3.7 Query accuracy probability

Query accuracy probability (P_A) [100] reflects the probability that a failure detector's output is correct at any random time. I computed the P_A associated with each detector in every experiment. Table 5.13 compiles the resulting values.

In a failure free context, all detectors exhibit a P_A above 0.9.Yet in this experiment as in all others, the P_A of SwimFD is notably lower than those of BertierFD and RepFD. SwimFD frequently suspects nodes because of the jitter induced by its suspicion list. It spends roughly 10% of its time suspecting nodes due to longer response times and pinging them to infirm the detection.

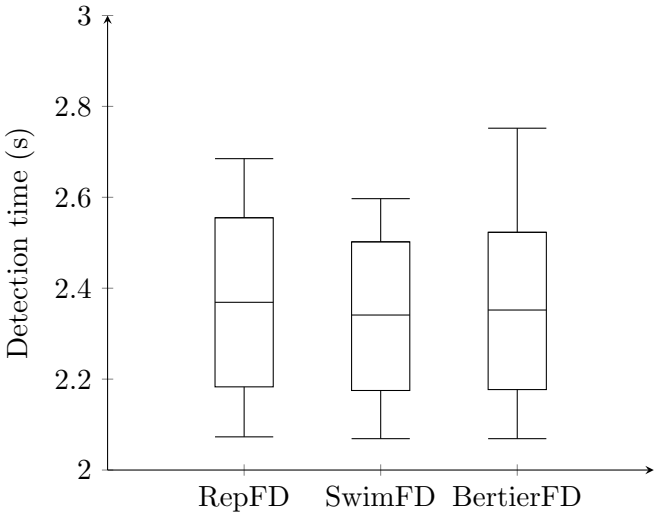


FIGURE 5.12: Overnet - Comparison of the detection times

Experiment	BertierFD	SwimFD	RepFD
No crash	0.999	0.904	0.998
Permanent crashes	0.999	0.886	0.997
Recovery	0.965	0.824	0.997
Frequent omissions	0.854	0.812	0.951

FIGURE 5.13: Query accuracy probability of the studied detectors

In the permanent crashes experiment, BertierFD exhibits a P_A that is consistent with its original crash detection results [97]. The accuracy of RepFD also remains constant in this experiment. SwimFD has to produce more messages when a crash occurs. This slightly impacts the whole system, lowering the detector's P_A down from 0.904 to 0.886: a 2% degradation.

In the crash/recovery experiment, both BertierFD and SwimFD incur a noticeable decrease of their P_A: a 3.5% and a 9.7% degradation respectively. As the delay of one detection is reproduced multiple times in the experiment, the accuracy of both state of the art detectors decreases. Note that the P_A for BertierFD remains above 0.96. My detector actually produces the best P_A value in this experiment. The responsiveness of the PID computation shines when it comes to addressing node recoveries repeatedly over time.

The FTA trace generating frequent omissions impacts strongly on both SwimFD and BertierFD, whereas it bears little impact on my detector. Their respective P_A results verify this observation: whereas my detector keeps a steady P_A value, strong dynamicity impedes the accuracy of BertierFD ($P_A = 0.854$). The accuracy of SwimFD remains the lowest but switching from infrequent crash/recoveries to frequent omissions degrades the accuracy of BertierFD the most: 11% degradation, whereas the accuracy of both

SwimFD and RepFD only incurs a 1% degradation. Bertier's RTT estimation does not converge fast enough, hence the important degradation of the accuracy of BertierFD during this experiment. Conversely, since SwimFD does not rely on past detections, the level of dynamicity of the system impacts less on the degradation of its accuracy.

5.3.8 Bandwidth usage

Experiment	BertierFD	SwimFD	RepFD
No crash	840.0	100.8	144.1
Permanent crashes	840.0	102.5	144.9
Recovery	840.0	109.2	145.7
Frequent omissions	840.0	336.2	252.0

FIGURE 5.14: Bandwidth consumption of the studied detectors in bytes/s per node

Another important metric for comparing failure detectors is their network cost. Subsection 5.2.3 assesses their respective message complexities, but I chose to measure the overhead generated by each detector in my experiments too. To assess this overhead, I measure bandwidth consumption by logging the number of messages sent and by assuming a default size of ping/heartbeat of 84 bytes: 20 bytes for IP headers, 8 bytes for the ICMP header, followed by the default 56 bytes of extra data as specified for Unix ping requests. In the case of RepFD, the size of heartbeats also includes updates of reputation values. The table of Figure 5.14 compares the network costs of the three studied detectors: it shows the average bandwidth consumption per node and per second in every scenario.

All the values in this table remain low by the standards of current broadband connection networks. However, it is important to remember that I target large and dynamic systems where network overloads are common. Moreover, some of the failures I aim to detect can be induced by such overloads. In this context, a lightweight protocol is paramount for failure detection.

As expected, the all to all heartbeat protocol of BertierFD is the most costly. It generates a constant overhead of nearly 1 kilobyte per second per node, regardless of the scenario.

Under favorable conditions, for instance a low frequency of failures, the optimistic approach of SwimFD is very cost-efficient. On the contrary, in a context of high omissions the suspicion mechanism of SwimFD triggers the degraded mode often. This generates a lot of network overhead, and the costs would rise even higher in a larger network.

RepFD also proves very cost-efficient. Its overhead is slightly higher than that of SwimFD when the frequency of failures is low. But RepFD reacts well to nodes with

very erratic heartbeat emissions: it consumes 25% less bandwidth than SwimFD in the Overnet experiment. Two factors contribute to this behavior: the low average degree of nodes imposed by my reputation system, and the fact that variations of the reputation values are piggybacked on heartbeats. An added advantage of this design is that it scales well.

5.3.9 PlanetLab experiment: introducing realistic jitter

In this section, I compare the performance of BertierFD, SwimFD, and RepFD in a realistic and jittery environment: the PlanetLab network [103]. The PlanetLab infrastructure holds no control over user processes: every user can launch any number of concurrent VMs. The resulting environment is well known as a challenging platform for any type of distributed algorithm. In particular, PlanetLab communications often incur heavy jitter, and a PlanetLab trace would allow a fair comparison of the detectors in a strongly degraded network.

To the best of my knowledge there is no database that provides reusable PlanetLab traces. I therefore built such a database, and made it freely available online [1]. My database stores the all-to-all network latencies incurred by 116 nodes spread all around Europe over 4 days. During those 4 days, every node pinged every other node every 500ms, and logged the latency associated with each ping. The 4-day duration might seem small but it is the longest period over which I monitored a significant number of nodes that remained stable enough for the results to be exploitable.

Figure 5.15 plots the median latency of all connections over four days. A relatively long phase of stability appears around the middle of day 2. This actually checks out as normal, as day 2 was a Sunday... Figure 5.16 complements the previous results by plotting the median standard deviation of these latencies: it represents the most common behavior of a PlanetLab connexion in terms of latency, even during stable periods, and shows that it is very erratic indeed.

Armed with my PlanetLab traces, I set up my detector comparison on top of 250 nodes: each of them replays a 1-hour sequence selected at random from the traces. To test quality of the detection in the case of crashes, I manually force 10 nodes to crash after 30 minutes and measure the average detection time resulting from these crashes.

The results of this experiment, summarized in Table 5.17, confirm my previous comparisons. BertierFD and SwimFD respectively produce false positives 57.8% and 61.8% of the time, while RepFD remains correct 96.7% of the time. In conclusion, RepFD

[1] https://pages.lip6.fr/Maxime.Veron/planetlab_pings.sql.bz2

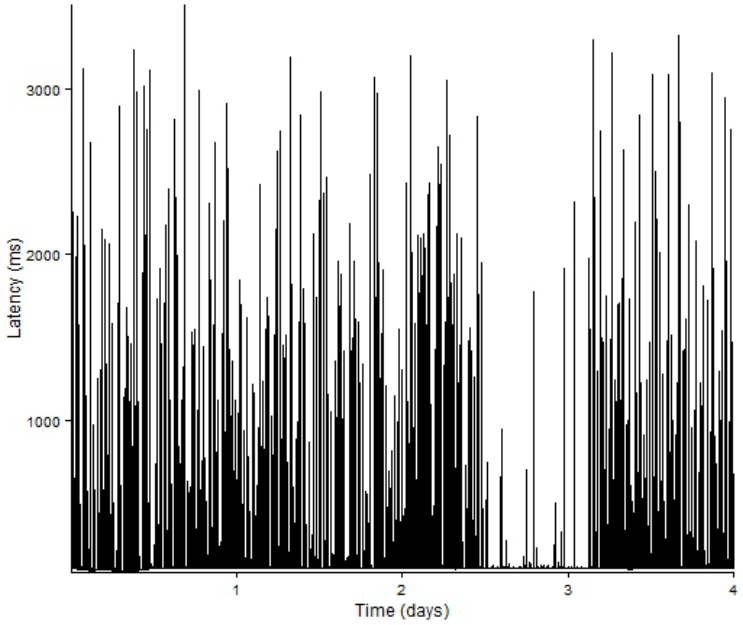


FIGURE 5.15: Median link latencies over four days

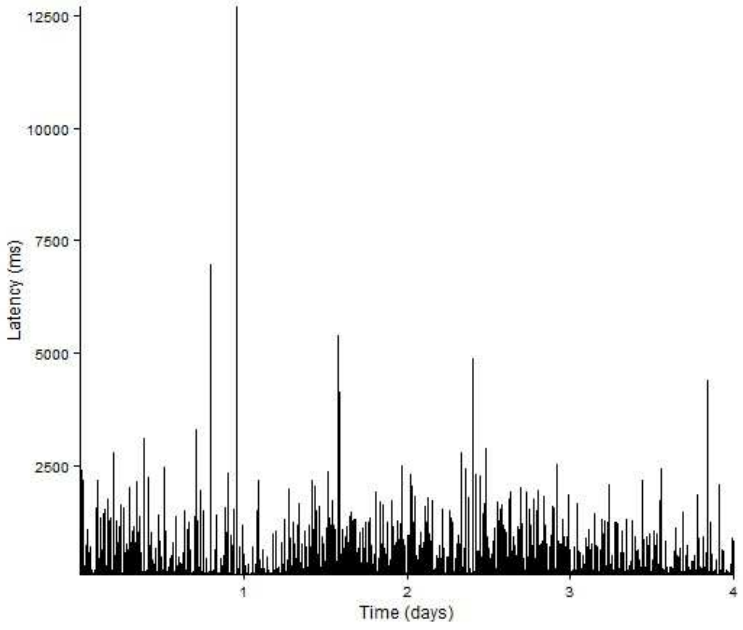


FIGURE 5.16: Median standard deviation of the latencies over the same period

FD	Detection time	PA
RepFD	2.6776	0.967
BertierFD	2.714	0.422
SwimFD	3.397	0.382

FIGURE 5.17: Performance comparison with PlanetLab traces

is much better than SwimFD and BertierFD at handling networks with high levels of jitter.

5.4 Related work

Since the introduction of the notion of unreliable failure detectors by Chandra and Toueg in 1996 [95] there have been many research efforts in this area.

A first class of detectors, such as Swim [99], is based on probing: nodes periodically probe their neighbors by sending ping messages. A second class relies on heartbeats: Bertier [97] and Chen [100] belong to this second class. Several efforts have been made towards scaling up failure detectors implementations. In [104], a hierarchical topology is used to reduced message complexity. Larrea *et al.* also aim to diminish the amount of exchanged information in order to scale up. To do so, they propose to use a logical ring to structure message exchanges [105]. Finally, Swim [99] scales by using a probabilistic approach: nodes randomly choose a subset of neighbors to probe.

In a recent work, Leners *et al.* propose Falcon [106]. It focuses on the fault detection speed and on the reliability of failure detectors: a process detected as faulty is never up (*i.e.,* always effectively faulty). Falcon may kill components to ensure this property. It is mainly designed to be used among processes within a same local area network (*e.g.,* a single data center).

All the failure detectors presented above classify processes either as correct or as faulty. Accrual failure detectors [96] associate a value with each process, which represents the risk that the process is indeed faulty.

My failure detector, based on a reputation mechanism, can rely on a more global (yet still fuzzy) view of the system than the state-of-the-art detectors. In this work I use my own reputation system presented in Section 4.4.1 to detect faults among nodes. However, other reputation systems such as [86] or systems with more advanced concepts like the ones described in [59, 107] could also act as failure detectors as long as the system is well tuned to detect faults. For short, the aim of this work is not to propose a novel reputation system, but to demonstrate that a reputation system can be used to build efficient failure detectors for large and dynamic networks.

5.5 Conclusion

This chapter presents an approach for implementing a reputation based failure detector that target large scale networks where dynamic reconfigurations are frequent. I show that a failure detector built on top of a reputation system improves accuracy and completeness while tolerating high levels of omissions. The exchange of subjective assessments among nodes leads to an efficient cooperation towards an accurate and up-to-date view of failures. Overall, a reputation-based detector combines reasonable detection times with good accuracy when the system runs well, and slows down the local detection just enough to prevent false positives when the system incurs frequent omissions.

Possible extensions of my work include a study on how well RepFD withstands network partitions, and a cooperative exploitation of the reputation assessments to identify simultaneous failures of multiple nodes.

Chapter 6

Conclusion

Contents

The scalability of massively multiplayer online games is a hot issue in the gaming industry. MMOGs aim at gathering an infinite number of players within the same virtual universe. Yet all existing MMOGs rely on centralized client/server architectures which impose a limit on the maximum number of players (avatars) and resources that can coexist in any given virtual universe.

6.1 Contributions

In order to improve the scalability of MMOGs services, I propose a set of contributions in three particular domain in relation with MMOGs: matchmaking, cheat detection, and more generically failure detection.

Matchmaking: I study multiple scalable matchmaking approaches that rely either on P2P architectures or centralized server regarding the playerbase size.

Throughout my study, I first of all revealed the flaws of current matchmaking solutions. In order to address this open issue, I describe some simple but efficient ways to make a matchmaking approaches scalable using the collected real gaming data.

From this experiment, I deduce target solutions for different services size by depicting both flaws and advantages of all different approaches. Changing the way algorithms are designed can easily improve the current approaches matching quality both in terms of matching response time and in terms of precision.

Cheat detection: I offer a scalable cheat detection protocol using a reputation system to assess nodes.

In this work, I propose a probabilistic solution based on a reputation system. It tests the behavior of nodes by submitting fake refereeing requests. By selecting those it identifies as trustworthy, it can referee real game actions. I ran experiments to test whether my solution is viable when real network conditions are involved. Experimental results included in this work show that it is possible to provide an efficient anti-cheat mechanism in peer to peer MMOGs without degrading their scalability.

Failure detection: I propose a generic failure detector, RepFD, allowing to detect nodes even under noisy network environments.

I show that a failure detector built on top of a reputation system achieves accuracy and completeness while tolerating high levels of omissions. The exchange of subjective assessments among nodes leads to an efficient cooperation towards an accurate and up-to-date view of node failures.

6.2 Future work

This thesis opens both direct and indirect paths in the domain of high scalability gaming. The three direct future work are in close relation to my three main contributions:

Matchmaking: This work offers many deeper studies of matchmaking and other gaming services using the database that we publicly offer. This study leads to the perspective that gaming matchmaking can scale using a P2P model, but does not need it as better implementations behave just as good if not better. In order to address current matchmaking issues, changes will have to be done either in the algorithmic way matchmaking is achieved or in the architecture itself, moving towards a high scale P2P solution.

Cheat detection: The solution can allow an extension of the cheat detection to more complex tasks such as gold farming detection, or behavioral patterns matching. As the reputation system is at the core of the system, it will also be relevant to analyze the importance and impact of the reputation computation in the whole system performance.

Failure detection: RepFD currently relies on a knowledge of the network quality in order to tune its PID parameters. A solution to this issue would be to propose an auto-tuning mechanism to RepFD. Using an online monitoring of the network it is possible to inject the past scenario and train RepFD parameters to converge towards the recent network performance. RepFD would therefor be able to be deployed in a

network without requiring a particular initial setup at the cost of a small startup phase of unreliability.

The indirect future work this thesis opens concerns a long term change of the gaming industry towards *cloud gaming*:

Cloud deployment is a step towards decentralization and a growing trend within the digital games industry [108, 109]. The work presented in this thesis focuses on a fully decentralized approach over P2P overlays, yet it also has potential as an extension to cloud deployment.

Cloud gaming extends the traditional C/S cluster-based approach and allows dynamic provisioning of server resources. Additionally, it can help alleviate the computing load on the client side by running all computations on cloud servers and streaming the resulting video feed [110].

The *cloud gaming* model has two main limitations. Firstly, dynamic resource provisioning is extremely complex [24] and can be very costly. Secondly, even current high-end clouds such as Amazon's EC2 cannot provide support for cloud gaming to 25% of their total end-user population because of latency issues [111]. In 2012 the combination of these issues almost lead to the bankruptcy of OnLive [112], one of the major cloud gaming pioneers.

Merging my P2P approaches with cloud deployment might mitigate these limitations. Exploiting freely available resources on peers increases the cost- efficiency of a hybrid architecture and eliminates the negative impact of overprovisioning. The delegation of computation loads to CPU-potent peers on the same LAN as CPU-deficient devices contributes to solving the latency issues. At the same time, adding cloud support to my solutions would definitely improve its performance and increase its reliability further, as cloud servers can constitute a core set of efficient and trusted nodes.

Bibliography

[1] J.L. Miller and J. Crowcroft. The near-term feasibility of P2P MMOG's. In *Network and Systems Support for Games (NetGames), 2010 9th Annual Workshop on*, pages 1 –6, November 2010. doi: 10.1109/NETGAMES.2010.5679578.

[2] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[3] World of warcraft subscribers per quarter, 2015. URL http://www.statista.com/statistics/276601/number-of-world-of-warcraft-subscribers-by-quarter/.

[4] World of tanks sets new guinness record, March 2013. URL http://worldoftanks.com/en/news/pc-browser/17/world-tanks-sets-new-guinness-world-record/.

[5] Diablo 3: gamers angry as servers collapse, May 2012. URL http://www.guardian.co.uk/technology/gamesblog/2012/may/15/diablo-3-gamers-angry-servers-collapse.

[6] 'world of warcraft: Warlords of draenor' release dogged by laggy servers, DDoS attack and WoW subreddit shutdown. URL http://www.techtimes.com/articles/20434/20141118/world-of-warcraft-warlords-of-draenor-release-dogged-by-laggy-servers-ddos-atta
htm.

[7] 5 lessons that mmos can learn from archeage, 2015. URL http://www.makeuseof.com/tag/5-lessons-mmos-can-learn-archeages-mistakes/.

[8] Diablo III players banned for botting. URL http://eu.battle.net/d3/en/forum/topic/6161067410.

[9] Recent actions against botting in hearthstone. URL http://us.battle.net/hearthstone/en/blog/16481223/recent-actions-against-botting-in-hearthstone-10-27-2014.

[10] Stefano Ferretti and Marco Roccetti. Game time modelling for cheating detection in P2P MOGs: a case study with a fast rate cheat. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-589-4. doi: 10.1145/1230040.1230045. URL http://doi.acm.org/10.1145/1230040.1230045.

[11] Steven Daniel Webb, Sieteng Soh, and William Lau. RACS: A Referee Anti-Cheat Scheme for P2P Gaming. In *Proc. of the Int. Conference on Network and Operating System Support for Digital Audio and Video (NOSSDAV'07)*, 2007.

[12] Is smite matchmaking flawed?, . URL http://www.reddit.com/r/Smite/comments/1zfp2b/trueskill_smites_match_making_system_is_it_flawed/.

[13] How smite matchmaking works, . URL http://www.vaultf4.com/threads/how-smite-matchmaking-works.2275/.

[14] Elth Ogston and Stamatis Vassiliadis. Local distributed agent matchmaking. In *In Proc. of the 9th International Conference on Cooperative Information Systems*, pages 67–79, 2001.

[15] Victor Shafran, Gal Kaminka, Sarit Kraus, and Claudia V. Goldman. Towards bidirectional distributed matchmaking. In *Proceedings of the 7th int. joint conference on Autonomous agents and multiagent systems*, AAMAS'08, pages 1437–1440, 2008. ISBN 978-0-9817381-2-3.

[16] Zili Zhang and Chengqi Zhang. An improvement to matchmaking algorithms for middle agents. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*, AAMAS '02, pages 1340–1347, New York, NY, USA, 2002. ACM. ISBN 1-58113-480-0. doi: 10.1145/545056.545129.

[17] Victor Shafran, Gal Kaminka, Sarit Kraus, and Claudia V. Goldman. Towards bidirectional distributed matchmaking. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 3*, AAMAS '08, pages 1437–1440, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-2-3. URL http://dl.acm.org/citation.cfm?id=1402821.1402892.

[18] Sharad Agarwal and Jacob R. Lorch. Matchmaking for online games and other latency-sensitive p2p systems. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 315–326, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-594-9. doi: 10.1145/1592568.1592605.

[19] Justin Manweiler, Sharad Agarwal, Ming Zhang, Romit Roy Choudhury, and Paramvir Bahl. Switchboard: A matchmaking system for multiplayer mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 71–84, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0643-0. doi: 10.1145/1999995.2000003.

[20] O. Delalleau, E. Contal, E. Thibodeau-Laufer, R.C. Ferrari, Y. Bengio, and F. Zhang. Beyond skill rating: Advanced matchmaking in ghost recon online. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(3):167–177, 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2188833.

[21] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 398–407, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4039-9. doi: 10.1109/CCGRID.2010.71. URL http://dx.doi.org/10.1109/CCGRID.2010.71.

[22] Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-589-4. doi: 10.1145/1230040.1230058.

[23] Davide Frey, Jerome Royan, Romain Piegay, Anne-Marie Kermarrec, Emmanuelle Anceaume, and Fabrice Le Fessant. Solipsis: A Decentralized Architecture for Virtual Environments. In *1st International Workshop on Massively Multiuser Virtual Environments*, Reno, NV, États-Unis, 2008.

[24] L. Ricci and E. Carlini. Distributed virtual environments: From client server to cloud and p2p architectures. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 8–17, 2012. doi: 10.1109/HPCSim.2012.6266885.

[25] M. Varvello, C. Diout, and E. Biersack. P2P second life: Experimental validation using kad. In *INFOCOM 2009, IEEE*, pages 1161 –1169, April 2009. doi: 10.1109/INFCOM.2009.5062029.

[26] Johanna Antila and Jani Lakkakorpi. On the effect of reduced quality of service in multiplayer On-Line games. *Int. J. Intell. Games & Simulation*, 2(2):89–95, 2003. URL http://dblp.uni-trier.de/rec/bibtex/journals/ijigs/AntilaL03.

[27] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in warcraft III. In *Proceedings of the 2nd*

workshop on Network and system support for games, NetGames '03, page 3"14, New York, NY, USA, 2003. ACM. ISBN 1-58113-734-6. doi: 10.1145/963900.963901. URL http://doi.acm.org/10.1145/963900.963901.

[28] Manish Joshi, Lall Manoj, O. Olugbara Oludayo, Michael M. Modiba, and C. Bhavsar Virendrakumar. Automated matchmaking of student skills and academic course requisites. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 514–519, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1185-4. doi: 10.1145/2381716.2381814.

[29] Manish Joshi, Virendrakumar C. Bhavsar, and Harold Boley. Matchmaking in p2p e-marketplaces: soft constraints and compromise matching. In *Proceedings of the 12th International Conference on Electronic Commerce: Roadmap for the Future of Electronic Business*, ICEC '10, pages 134–140, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-1427-5. doi: 10.1145/2389376.2389395. URL http://doi.acm.org/10.1145/2389376.2389395.

[30] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill^{TM}: A bayesian skill rating system. In *Advances in Neural Information Processing Systems*, pages 569–576, 2006.

[31] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in warcraft iii. In *Proceedings of the 2nd workshop on Network and system support for games*, NetGames '03, pages 3–14, New York, NY, USA, 2003. ACM. ISBN 1-58113-734-6. doi: 10.1145/963900.963901.

[32] G. Armitage. An experimental estimation of latency sensitivity in multiplayer quake 3. In *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, pages 137–141, 2003.

[33] Sebastian Zander, Ian Leeder, and Grenville Armitage. Achieving fairness in multiplayer network games through automated latency balancing. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, ACE '05, pages 117–124, New York, NY, USA, 2005. ACM. ISBN 1-59593-110-4. doi: 10.1145/1178477.1178493.

[34] Arpad E. Elo. *The rating of chessplayers, past and present*. Arco Pub., New York, 1978. ISBN 0668047216 9780668047210.

[35] MatchMaking in league of legends, 2013. URL http://euw.leagueoflegends.com/learn/gameplay/matchmaking.

[36] League of legends community - general discussion - matchmaking changes [1-16-2013]. URL http://na.leagueoflegends.com/board/showthread.php?t=3016615.

[37] Zynox. FunnyPullack. http://www.zynox.net/category/games/league-of-legends/funnypullback/. URL http://www.zynox.net/category/games/league-of-legends/funnypullback/.

[38] Blizzard Entertainment:Diablo 2. http://us.blizzard.com/en-us/games/d2/index.html, . URL http://us.blizzard.com/en-us/games/d2/index.html.

[39] Athene abusing matching making system to get Diamond. URL http://forums.na.leagueoflegends.com/board/showthread.php?t=3274038.

[40] Lag switch in FPS. URL http://www.examiner.com/article/how-cheaters-cheat-lag-switches.

[41] Nintendo can brick your 3DS remotely if modified. URL http://www.mmofringe.com/forum/8-News-Insights-and-Theories/10979-nintendo-can-brick-your-3ds-remotely-if-modified.

[42] 25C3: console hacking 2008: Wii fail. URL http://events.ccc.de/congress/2008/Fahrplan/events/2799.en.html.

[43] 27C3: console hacking 2010. URL http://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html.

[44] Wu-chang Feng, Ed Kaiser, and Travis Schluessler. Stealth measurements for cheat detection in on-line games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '08, page 15"20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-132-3. doi: 10.1145/1517494.1517497. URL http://doi.acm.org/10.1145/1517494.1517497.

[45] Travis Schluessler, Stephen Goglin, and Erik Johnson. Is a bot at the controls?: Detecting input data attacks. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, NetGames '07, page 1"6, New York, NY, USA, 2007. ACM. ISBN 978-0-9804460-0-5. doi: 10.1145/1326257.1326258. URL http://doi.acm.org/10.1145/1326257.1326258.

[46] Warden : Blizzard cheat detection tool. URL http://en.wikipedia.org/wiki/Warden_%28software%29.

[47] Sam Moffatt, Akshay Dua, and Wu-chang Feng. SpotCheck: an efficient defense against information exposure cheats. In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*, NetGames '11, page 8:1“8:6, Piscataway, NJ, USA, 2011. IEEE Press. ISBN 978-1-4577-1934-9. URL http://dl.acm.org/citation.cfm?id=2157848.2157857.

[48] Josh Goodman and Clark Verbrugge. A peer auditing scheme for cheat elimination in mmogs. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '08, pages 9–14, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-132-3. doi: 10.1145/1517494.1517496.

[49] A. Yahyavi, K. Huguenin, J. Gascon-Samson, J. Kienzle, and B. Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 134–144, July 2013. doi: 10.1109/ICDCS.2013.62.

[50] Jared Jardine and Daniel Zappala. A hybrid architecture for massively multiplayer online games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '08, page 60“65, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-132-3. doi: 10.1145/1517494.1517507. URL http://doi.acm.org/10.1145/1517494.1517507.

[51] J.-Y. Cai, A. Nerurkar, and Min-You Wu. Making benchmarks uncheatable. In *Computer Performance and Dependability Symposium, 1998. IPDS '98. Proceedings. IEEE International*, pages 216–226, Sep 1998.

[52] Daniel Pittman and Chris GauthierDickey. Cheat-proof peer-to-peer trading card games. In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*, NetGames '11, page 9:1“9:6, Piscataway, NJ, USA, 2011. IEEE Press. ISBN 978-1-4577-1934-9. URL http://dl.acm.org/citation.cfm?id=2157848.2157858.

[53] Kuan-Ta Chen, Hsing-Kuo Kenneth Pao, and Hong-Chung Chang. Game bot identification based on manifold learning. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '08, page 21“26, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-132-3. doi: 10.1145/1517494.1517498. URL http://doi.acm.org/10.1145/1517494.1517498.

[54] Kusno Prasetya and Zheng da Wu. Artificial neural network for bot detection system in MMOGs. In *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games*, NetGames '10, page 16:1“16:2, Piscataway, NJ, USA, 2010. IEEE Press. ISBN 978-1-4244-8355-6. URL http://dl.acm.org/citation.cfm?id=1944796.1944812.

[55] Christian Mönch, Gisle Grimen, and Roger Midtstraum. Protecting online games against cheating. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-589-4. doi: 10.1145/1230040.1230087. URL http://doi.acm.org/10.1145/1230040.1230087.

[56] Margaret DeLap, Björn Knutsson, Honghui Lu, Oleg Sokolsky, Usa Sammapun, Insup Lee, and Christos Tsarouchis. Is runtime verification applicable to cheat detection? In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, page 134"138, New York, NY, USA, 2004. ACM. ISBN 1-58113-942-X. doi: 10.1145/1016540.1016553. URL http://doi.acm.org/10.1145/1016540.1016553.

[57] F. Tegeler and Xiaoming Fu. SybilConf: computational puzzles for confining sybil attacks. In *INFOCOM IEEE Conference on Computer Communications Workshops , 2010*, pages 1 –2, March 2010. doi: 10.1109/INFCOMW.2010.5466685.

[58] E. Rosas, O. Marin, and X. Bonnaire. CORPS: Building a Community Of Reputable PeerS in Distributed Hash Tables. *The Computer Journal*, 54(10):1721–1735, September 2011. ISSN 0010-4620, 1460-2067. doi: 10.1093/comjnl/bxr087.

[59] Audun Jøsang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644, March 2007.

[60] A.G.P. Rahbar and O. Yang. PowerTrust: A Robust and Scalable Reputation System for Trusted Peer-to-Peer Computing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(4):460 –473, April 2007. ISSN 1045-9219. doi: 10.1109/TPDS.2007.1021.

[61] Daniel Houser and John Wooders. Reputation in auctions: Theory, and evidence from eBay. *SSRN eLibrary*. URL http://papers.ssrn.com/sol3/papers.cfm?abstract_id=899187.

[62] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Commun. ACM*, 43(12):45"48, 2000. ISSN 0001-0782. doi: 10.1145/355112.355122. URL http://doi.acm.org/10.1145/355112.355122.

[63] Edward Kaiser and Wu-chang Feng. PlayerRating: a reputation system for multiplayer online games. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, NetGames '09, page 8:1"8:6, Piscataway, NJ, USA, 2009. IEEE Press. ISBN 978-1-4244-5605-5. URL http://dl.acm.org/citation.cfm?id=1837164.1837176.

[64] Xavier Bonnaire and Erika Rosas. WTR: a reputation metric for distributed hash tables based on a risk and credibility factor. *Journal of Computer Science and Technology*, 24(5):844–854, September 2009. ISSN 1000-9000, 1860-4749. doi: 10.1007/s11390-009-9276-6. URL http://hal.inria.fr/inria-00627507.

[65] Audun Jøsang, Xixi Luo, and Xiaowu Chen. Continuous ratings in discrete bayesian reputation systems. In Yücel Karabulut, John Mitchell, Peter Herrmann, and ChristianDamsgaard Jensen, editors, *Trust Management II*, volume 263 of *IFIP – The International Federation for Information Processing*, pages 151–166. Springer US, 2008. ISBN 978-0-387-09427-4. doi: 10.1007/978-0-387-09428-1_10. URL http://dx.doi.org/10.1007/978-0-387-09428-1_10.

[66] Matúš Medo and Joseph Rushton Wakeling. The effect of discrete vs. continuous-valued ratings on reputation and ranking systems. *EPL (Europhysics Letters)*, 91 (4):48004, 2010. URL http://stacks.iop.org/0295-5075/91/i=4/a=48004.

[67] Chrysanthos Dellarocas. Immunizing online reputation reporting systems against unfair ratings and discriminatory behavior. In *Proceedings of the 2Nd ACM Conference on Electronic Commerce*, EC '00, pages 150–157, New York, NY, USA, 2000. ACM. ISBN 1-58113-272-7. doi: 10.1145/352871.352889. URL http://doi.acm.org/10.1145/352871.352889.

[68] Andrew Whitby, Audun Jøsang, and Jadwiga Indulska. Filtering out unfair ratings in bayesian reputation systems. 2004.

[69] Henry S Teng, Kaihu Chen, and Stephen C Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 278–284. IEEE, 1990.

[70] Anup K Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Workshop on Intrusion Detection and Network Monitoring*, volume 51462, 1999.

[71] Rachna Dhamija, J Doug Tygar, and Marti Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590. ACM, 2006.

[72] Min Wu, Robert C Miller, and Simson L Garfinkel. Do security toolbars actually prevent phishing attacks? In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 601–610. ACM, 2006.

[73] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1065–1074. ACM, 2008.

[74] Nikita Borisov. Computational puzzles as sybil defenses. In *Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, P2P '06, page 171"176, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2679-9. doi: 10.1109/P2P.2006.10. URL http://dx.doi.org/10.1109/P2P.2006.10.

[75] League of legends game traces database, 2013. URL http://pagesperso-systeme.lip6.fr/maxime.veron/examples.html.

[76] Maxime Véron, Olivier Marin, and Sébastien Monnet. Matchmaking in multi-player on-line games: Studying user traces to improve the user experience. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, NOSSDAV '14, pages 7:7–7:12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2706-0. doi: 10.1145/2578260.2578265. URL http://doi.acm.org/10.1145/2578260.2578265.

[77] lolrtmpsclient - a lightweight rtmp client in java for communicating with league of legends, 2013. URL https://code.google.com/p/lolrtmpsclient/.

[78] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. How sensitive are online gamers to network quality? *Commun. ACM*, 49(11):34–38, November 2006. ISSN 0001-0782. doi: 10.1145/1167838.1167859.

[79] ILOG CPLEX. High-performance software for mathematical programming and optimization, 2005.

[80] Alberto Montresor and Márk Jelasity. PeerSim: A Scalable P2P Simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, sep 2009.

[81] Blizzard Admits Diablo 3 Item Duping is why asian server was shut down, June 2012. URL http://www.cinemablend.com/games/Blizzard-Admits-Diablo-3-Item-Duping-Why-Asia-Server-Was-Shutdown-43472.html.

[82] Blizzard Seeking Hacker Behind WoW Insta-Kill Massacre. http://www.tomshardware.com/news/World-ofWarcraft-Aura-of-Gold-MMORPG-Insta-kill-exploit, . URL http://www.tomshardware.com/news/World-ofWarcraft-Aura-of-Gold-MMORPG-Insta-kill-exploit,18219.html.

[83] Maxime Véron, Olivier Marin, Sébastien Monnet, and Zahia Guessoum. Towards a scalable refereeing system for online gaming. *Multimedia Systems*, pages 1–15, 2014. ISSN 0942-4962. URL http://dx.doi.org/10.1007/s00530-014-0358-0.

[84] Stephen M. Specht and Ruby B. Lee. Distributed denial of service: taxonomies of attacks, tools and countermeasures. In *Proc. of the Int. Workshop on Security in Parallel and Distributed Systems*, pages 543–550, 2004.

[85] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 389–400, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-175-0. doi: 10.1145/1402958.1403002.

[86] Mudhakar Srivatsa, Li Xiong, and Ling Liu. TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 422–431, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9. doi: 10.1145/1060745.1060808.

[87] World Of Warcraft Servers average concurrent players, . URL http://www.warcraftrealms.com/activity.php?serverid=-1.

[88] World Of Warcraft Servers player capacity, . URL http://www.warcraftrealms.com/realmstats.php?sort=Overall.

[89] Kuan-Ta Chen, Polly Huang, Chun-Ying Huang, and Chin-Laung Lei. Game traffic analysis: an mmorpg perspective. In *Proceedings of the international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '05, pages 19–24, New York, NY, USA, 2005. ACM. ISBN 1-58113-987-X.

[90] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003. ISSN 0146-4833. doi: 10.1145/956993.956995.

[91] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9492-5. doi: 10.1109/GRID.2005.1542730.

[92] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, May 2002. ISSN 0018-9340. doi: 10.1109/TC.2002.1004595. URL http://dx.doi.org/10.1109/TC.2002.1004595.

[93] Steven Daniel Webb, Sieteng Soh, and William Lau. RACS: A Referee Anti-Cheat Scheme for P2P Gaming. In *Proc. of the Int. Conference on Network and Operating System Support for Digital Audio and Video (NOSSDAV'07)*, 2007.

[94] Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-589-4. doi: 10.1145/1230040.1230058. URL http://doi.acm.org/10.1145/1230040.1230058.

[95] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[96] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The phi; accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems. SRDS 2004.*, pages 66–78, Oct 2004.

[97] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proc. of the int. conf. on Dependable Systems and Networks, 2002. DSN 2002.*, pages 354–363, 2002. doi: 10.1109/DSN.2002.1028920.

[98] A. Tomsic, P. Sens, J. Garcia, L. Arantes, and J. Sopena. 2w-fd: A failure detector algorithm with qos. In *International Parallel and Distributed Systems*, 2015.

[99] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 303–312, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1597-5.

[100] Wei Chen, S. Toueg, and M.K. Aguilera. On the quality of service of failure detectors. *Computers, IEEE Transactions on*, 51(5):561–580, May 2002.

[101] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, August 1988. ISSN 0146-4833.

[102] Maxime Véron, Olivier Marin, and Sébastien Monnet. Matchmaking in multi-player on-line games: Studying user traces to improve the user experience. In

Proceedings of Network and Operating System Support on Digital Audio and Video Workshop, NOSSDAV '14, pages 7:7–7:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2706-0.

[103] Larry Peterson, Steve Muir, Timothy Roscoe, and Aaron Klingaman. PlanetLab Architecture: An Overview. Technical Report PDN–06–031, PlanetLab Consortium, May 2006.

[104] Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of a hierarchical failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, pages 635–644, San Francisco, CA, june 2003.

[105] Mikel Larrea, Sergio Arévalo, and Antonio Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of Distributed Computing, 13th International Symposium 1999 (DISC '99)*, volume 1693 of *Lecture Notes in Computer Science*, pages 34–48, Bratislava, Slavak Republic, September 1999. Springer.

[106] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043583. URL http://doi.acm.org/10.1145/2043556.2043583.

[107] Amir Ban and Nati Linial. The dynamics of reputation systems. In *Proceedings of the 13th Conference on Theoretical Aspects of Rationality and Knowledge*, TARK XIII, pages 91–100, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0707-9.

[108] Onlive - The leader in cloud gaming, . URL http://www.onlive.com/.

[109] Nvidia Grid - cloud gaming, . URL http://blogs.nvidia.com/blog/2013/08/30/best-in-show/.

[110] R. Shea, Jiangchuan Liu, E.C.-H. Ngai, and Yong Cui. Cloud gaming: architecture and performance. *Network, IEEE*, 27(4):–, 2013. ISSN 0890-8044. doi: 10.1109/MNET.2013.6574660.

[111] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm in cloud gaming: a measurement study on cloud to end-user latency. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, NetGames '12, pages 2:1–2:6, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-4578-1.

[112] Onlive goes bankrupt. URL http://www.theverge.com/2012/8/28/3274739/onlive-report.